WiPiEC Journal

Works in Progress in Embedded Computing Journal

Special Issue on selected papers of
Works in Progress Session within 29th Euromicro Conference on Digital System Design (DSD) and 52th Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA), Krakow, Poland, Sept. 02th – 04th, 2026

August 2026 · Volume 12 · Issue 2 · ISSN 2980-7298

https://wipiec.digitalheritage.me

Message from the Guest Editors

It is our great pleasure to serve as the Guest Editors of this Special Edition of the WiPiEC Journal, featuring selected papers from the Work in Progress (WiP) Session. This session was held within the framework of the 29th Euromicro Conference on Digital System Design (DSD 2026) and the 52nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2026), held in Krakow, Poland, from September 02–05, 2026.

This marks the 4th WiPiEC edition dedicated to the DSD/SEAA conferences, which represent some of the longest-standing traditions in the fields of electronics, computer science, and the modern technologies derived from them. This year, we are pleased to present 15 selected papers covering a broad range of advanced thematic areas.

The success of the WiP Session would not have been possible without the valuable contributions of the authors, the enthusiastic support of the Program Committees, and the exceptional dedication of the MECOnet, MANT, and EUROMICRO staff. We extend our sincere gratitude to Jovan Djurković for his outstanding work on the technical editing and preparation of this volume. As in previous years, the Works in Progress in Embedded Computing (WiPiEC) Journal proudly publishes these selected papers in a fully open-access format. The quality of the contributions is exemplary, with numerous papers originating from leading universities and prominent research centers worldwide.

We are confident that the WiP Session will continue to expand and attract growing interest, particularly among young researchers who play a vital role in shaping the future of our scientific community. Over time, WiPiEC has established itself as a reputable journal, indexed in leading reference databases and repositories within the field of computer science.Welcome to DSD 2026, SEAA 2026, and the Work in Progress (WiP) Session. We offer our warmest congratulations to all the authors whose papers have been selected for publication in the WiPiEC Journal, Vol. 13, No. 2.

Sincerely,

Guest Editors:

Prof. Dr Radovan Stojanović
University of Montenegro, Montenegro

Prof. Dr Andrej Škraba
University of Maribor, Slovenia

Contents

Federico Nicolás Peccia, Avik Bhatnagar and Oliver Bringmann
Autotuned Distribution of Multi-DNN Workloads on Multi-Accelerator SoCs 1

Ludovic Claudepierre, Edna Rocio Ferrucho-Alvarez, Laurent Le Brizoual, Laurent Pichon
Clock-glitch Fault Characterization by Timing Comparison with Laser Fault Injection..... 9

Ahmet Zahit Can and Erkan Uslu
Dolunay: Architectural Support for Independent Thread Scheduling in a RISC-V SIMT Accelerator 17

Raphaël Wintersdorff, Renaud Pacalet and Laurent Sauvage
Operation Count as a Performance Predictor: An Empirical Study of Lightweight AEAD Schemes..... 25

João Antônio Temochko Andre and Alexandre Beletti Ferreira
Instruction Flow Amplifier: A Novel Architecture for In-Order Processor Front-End Optimization 33

Łukasz Drzensla, Sebastian Koryciak, Olksandr Savchenko and Paweł Russek
Study of PicoTDC Performance for FIT Detector Upgrade in ALICE Experiment at CERN.... 40

Jan Ber
Design and Implementation of Three-stage RISC-V Microprocessor for Education 46

João Maia Aguiar and José Machado da Silva
A Low-Power ADC-Free Resistive Sensor Readout Circuit for μC-Based Microsystems 51

Matthias Dippold, Sofia Maragkou, Matthias Wess, Thilo Sauter, Grigorios Chrysos and Sotiris Ioannidis
A Dual-Head Model for Host based Intrusion Detection on System-Call traces 57

Federico Buccellato, Luca Mannini, Corrado De Sio
A Compiler-Aware Framework for Partitioned Neural Network Inference on FPGA DPUs..... 65

Giorgia Paisi, Francesca Arcelli Fontana and Bartosz Walter
An Exploratory Study on Code Smells Detection and Refactoring using LLMs 73

Vadim Peczyński and Joanna Szłapczyńska
Microservice Decomposition using Many-Objective Optimization 80

Norbert Fijałek, Ilona Bluemke and Piotr Gawrysiak
Knowledge Retrieval Architectures for AI-Assisted Software Development: From Static Context to Autonomous Development Agents 90

Stefan Biffl, Tobias Bein, Sebastian Kropatschek, Kristof Meixner and Arndt Lüder
SIAM DSL: Accessible Structured Text for Bridging the Architecture-Impact Gap in Cyber-Physical Production Systems 98

Nadeem Abbas and Nazia Shahzadi
Large Language Models for Software Architecture Design Support in Self-Adaptive Systems: Early Insights from an Exploratory Systematic Review 106

Autotuned Distribution of Multi-DNN Workloads on Multi-Accelerator SoCs

Federico Nicolás Peccia, Avik Bhatnagar, Oliver Bringmann
FZI Research Center for Information Technology, University of Tübingen
Germany
peccia@fzi.de, bhatnagar@fzi.de, oliver.bringmann@uni-tuebingen.de

Abstract— **Many machine learning applications require heterogeneous Deep Neural Networks (DNNs) to work collaboratively. Although several works have focused on how to serve these systems using cloud solutions, less attention has been paid to the edge scenarios. Particularly, coordinating heterogeneous AI workloads across a custom application-specific System-on-Chip (SoC) with multiple accelerators presents significant challenges. This work first demonstrates how to implement such a baseline system using a SoC generator framework, performs an ablation study prototyping different versions on an FPGA, details how an RTOS can be used to achieve model parallelism on multiple accelerators, and identifies gaps and limitations by executing a multi-DNN autonomous driving application. To improve the utilization of the system and increase throughput, we propose a method to distribute the execution of individual layers across accelerators. Instead of partitioning all layers in the same manner and statically allocating them at compile time, we select the ideal partitioning for each one during compilation using an autotuning process, and then dynamically assign them to the available accelerators during runtime. We analyze the variability in layer execution in a system with multiple accelerators and use this information to guide the runtime allocation of partitions. We demonstrate that our method achieves a mean 29 % and 40 % improvement in accelerator utilization and throughput over the model parallelism baseline, and a 10 % and 9 % improvement over a round-robin runtime distribution of partitions.**

Multi-tenant DNN, FPGA, RTOS, SoC

I. INTRODUCTION

Machine learning applications are becoming increasingly complex, with some of them composed of multiple heterogeneous DNNs, each one trained for a particular task. Some edge multi-DNN application examples are autonomous driving [1] (object detection, tracking, and trajectory prediction) or AR/VR [2] (depth estimation and face recognition).

Multiple works have focused on INFerence-as-a-Service (INFaaS) solutions, where multi-accelerator SoCs are used as backend components of cloud inference systems [3],[4]. These works focus on optimizing cloud-specific metrics like Quality of Service (QoS) or service level agreement (SLA) satisfaction rates. Works that focus on edge solutions usually use commercial-off-the-shelf (COTS) embedded GPUs to deploy multi-DNN systems [5]}.

But the orchestration of heterogeneous DNNs on multi-accelerator custom application-specific SoCs has not been explored as thoroughly. Moreover, both edge and cloud works have only focused on evaluation at the simulator level and ignored implementation challenges. For example, the computational complexity of some of the runtime scheduling algorithms proposed by cloud solutions to dynamically allocate and distribute the execution of layers across the available accelerators typically prevents their deployment on the edge.

We present how such a multi-DNN multi-accelerator custom application-specific SoC for edge applications can be realized using an open-source hardware (HW) generation workflow based on Chipyard [6] and the Gemmini accelerator [7]. We show how the Zephyr RTOS [8] can be configured to achieve model parallelism using a reference application for autonomous driving requiring two heterogeneous DNNs. By executing the multi-DNN system on different HW versions, we identify that model parallelism alone limits the utilization of the available accelerators, resulting in idle resources.

To increase the utilization of the SoC's accelerators, this work proposes to parallelize the execution of individual DNN layers across multiple accelerators. But instead of relying on heuristics to decide how to partition each layer, we propose a workflow based around the TVM Deep Learning compiler [9] to automatically search for the best partitioning configuration for each layer at compile time (autotuning). During runtime, a lightweight allocation algorithm dynamically selects to which accelerators to assign the partitions of each layer. We show how the latency of a layer varies when multiple accelerators share the same memory subsystem, and how this distribution of possible latencies needs to be taken into account when calculating the pending load of each accelerator.

Our method achieves a mean 29 % improvement in accelerator utilization when compared against the model parallelism baseline. Compared to distributing the layer partitions in a round-robin fashion, our method achieves a mean 10 % increase in accelerator utilization. Additionally, the throughput of the partitioned DNN is increased by 40 % and 9 % against the model parallelism baseline and round-robin distribution.

This research was funded by the German Federal Ministry of Research, Technology, and Space (BMFTR), and the EU ChipsJU as part of the project RIGOLETTO under Grant 16MEE0547 and 101194371, respectively.

Manuscript received May 06, 2026; revised July 15, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.134

The rest of this paper is organized as follows. First, Section II discusses previous cloud and edge works. Then, Section III shows how such a multi-DNN multi-accelerator custom edge system can be realized using an RTOS and provides the ablation study measurements showcasing its limitations. Section IV presents our proposal to better utilize the idle accelerators of the system, which is then evaluated in Section V. Finally, Section VI concludes the work.

II. Related Work

Initial works on the deployment of multi-DNN workloads focused on interleaving the execution of DNNs on the same accelerator (*temporal* multi-tenancy). For example, AI-MT [10] proposed a HW scheduler to achieve this. On the other hand, Layerweaver [11] implemented a software (SW) scheduler to achieve the same goal. [12] used a preemption mechanism for accelerators and then proposed PREMA, an algorithm that leverages this preemption feature to manage DNN workloads with different priorities.

Other works focused on *spatial* multi-tenancy, where different workloads are assigned to different accelerators, adding parallelism to the execution. Planaria [13] proposed a HW design that allows the creation of sub-accelerators dynamically, and used this dynamic reconfiguration to run different workloads in parallel. Herald [14] proposed a way to orchestrate DNNs on heterogeneous dataflow accelerators by assigning each layer to the accelerator best suited for it. MAGMA [15], [17] and COMB [18] use Genetic Algorithms (GA) to find an optimized mapping for a set of DNNs on multiple accelerators sharing the same memory interface and take into account bandwidth allocation during the optimization. Layer-Puzzle [4] uses a dynamic scheduler framework to parallelize the execution of individual layers. CD-MSA [3] uses a deadline-aware scheduler to improve Quality of Service (QoS) guarantees for cloud use cases. Finally, works like Versa-DNN [19] proposed versatile HW accelerators able to reconfigure themselves into sub-accelerators during runtime, and present the appropriate algorithms to utilize them.

Table I compares our work against the literature. **Parallelism level** refers to the scheduling granularity: allocating entire *layers* to each accelerator, or partitioning the execution of a layer and distributing the resulting *tensor* operations. **Allocation** references the runtime behavior: some works decide the accelerator assignment offline (*static*); others do so during runtime (*dynamic*). The target **Application** modifies design decisions, particularly in dynamic allocation works: given its complexity, most dynamic algorithms developed for INFaaS would not be suitable for edge cases.

Other works have also leveraged TVM to distribute DNN execution across multiple accelerators in custom SoCs, such as MATCH [20]. However, our approach differs fundamentally in three key aspects: 1) while MATCH distributes the execution of complete layers, our work partitions individual layers into finer-grained tensor operations and distributes them, 2) MATCH performs accelerator mapping at compile time, whereas our distribution decisions occur at runtime, and 3) MATCH supports single DNN execution, while our runtime enables the concurrent execution of multiple DNNs.

TABLE I. Our work compared against others

	Year	Parallelism level	Allocation	Application	Evaluated on
PREMA [12]	2020	-	-	INFaaS	Simulator
AI-MT [10]	2020	-	-	INFaaS	Simulator
Planaria [13]	2020	Tensor	Dynamic	INFaaS	Simulator
Herald [14]	2021	Tensor	Static	Both	Simulator
Layerweaver [11]	2021	Layer	Static	INFaaS	Simulator
MAGMA [15]	2022	Layer	Static	INFaaS	Simulator
H2H [16]	2022	Layer	Static	INFaaS	Simulator
Layer-Puzzle [4]	2023	Tensor	Dynamic	INFaaS	Simulator
[17]	2023	Tensor	Static	Edge	Simulator
CD-MSA [3]	2023	Tensor	Dynamic	INFaaS	Simulator
COMB [18]	2023	Layer	Static	Edge	Simulator
Versa-DNN [19]	2024	Tensor	Dynamic	INFaaS	Simulator
Ours	**2026**	**Tensor**	**Dynamic**	**Edge**	**FPGA**

III. Baseline multi-DNN multi-accelerator system implementation

A generic multi-DNN multi-accelerator system is comprised of a set of M accelerators $A = \{acc_0, \dots, acc_{M-1}\}$, and executes N DNN workloads $G = \{G_0, \dots, G_{N-1}\}$. During runtime, requests are received for each DNN workload, and the system orchestrates the execution of each one of them on the available accelerators. The request arrival rate is application-specific: it could be periodic (like images coming from a camera stream) or batched (a one-time batch of requests). From these definitions, several challenges arise.

First of all, the number and size of the available accelerators need to be defined. These are usually constrained by area or power requirements, but the application to be executed on them should also be taken into account. For example, an application requiring high parallelism may benefit more from multiple smaller accelerators. On the other hand, an application focused on throughput may already fulfil its requirements with only one powerful accelerator.

Second, some access control mechanism needs to be provided to avoid concurrent SW access to the same HW resource. This can be realised using an RTOS resource, like a mutex, to block each accelerator until it finishes its computation. There are two possible blocking granularities. In the first one (model-wise), each request blocks the accelerator until it executes all its layers. From a request point of view, this is the *non-preemptible* case: once a request starts to execute on one accelerator, the RTOS cannot select another request to run on the same accelerator. In the second option (layer-wise), each request blocks the accelerator only until it finishes the execution of its current layer. This is the *preemptible* case: the RTOS is allowed to interleave the execution of layers from different requests on the same accelerator.

A third execution granularity exists, where the execution of a layer on an accelerator is interrupted before finishing its execution, in order to execute a new layer on it. But this would require storing the current accelerator state and then retrieving it once the layer can continue its execution. If a HW mechanism is not available to take care of this, it can result in a considerable

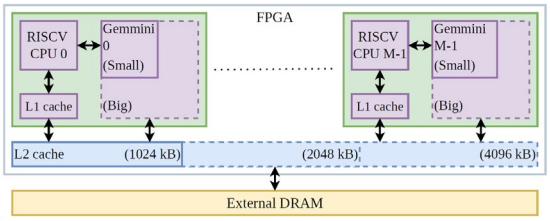


Figure 1 Multi-accelerator SoC under analysis

time overhead. So we focus our analysis only on the first two granularities.

Finally, once a new request arrives, the system needs to select which of the available accelerators should execute the model for this particular request. New requests should ideally be assigned to idle (or soon-to-be) accelerators.

A. *Selected application*

We choose a 3D object detection application to demonstrate a multi-DNN workload use case, where the 3D object properties are estimated from a single camera image by leveraging deep learning and geometric constraints [1]. Our implementation of the paper consists of two DNNs: a YOLOv7-tiny model to obtain all 2D object detections within an image in a single-shot operation, and a VGG-19-based DNN that regresses the 3D object parameters *for each* detected object.

Since the number of times the VGG model needs to be executed is dependent on the number of detections found by the YOLO model, the exact number of VGG requests is **unknown at compile time**. Therefore, we cannot statically define where each VGG model request needs to be executed. This is why some method is needed during runtime to dynamically assign requests to different accelerators.

B. *Hardware/software baseline*

Our SoC (Fig. 1) is built around the Chipyard generator framework. We implement *M* instances of the Gemmini accelerator, which contains a parametrized systolic array of DIMxDIM Processing Elements (PEs). This accelerator was selected as its applicability for real-time applications was already analyzed in [21]. The Zephyr RTOS is used to orchestrate the execution of the DNNs in *G*. To map the DNN operations onto the accelerator, we used the open-source integration in [22].

We configure the RTOS with a set of threads and use semaphores to exchange data between them. One thread is instantiated for each combination of workload in *G* and accelerator in *A*. Threads can only execute on the accelerator they are assigned to. Each new workload request is signalled through a semaphore to all the threads that can manage the corresponding model. This results in *automatic* model parallelism: if multiple requests are received, no matter the workload, they can be processed by different threads and therefore be executed on different accelerators.

Fig. 2 shows how these SW aspects modify the execution when one or two accelerators are available, and the metrics that

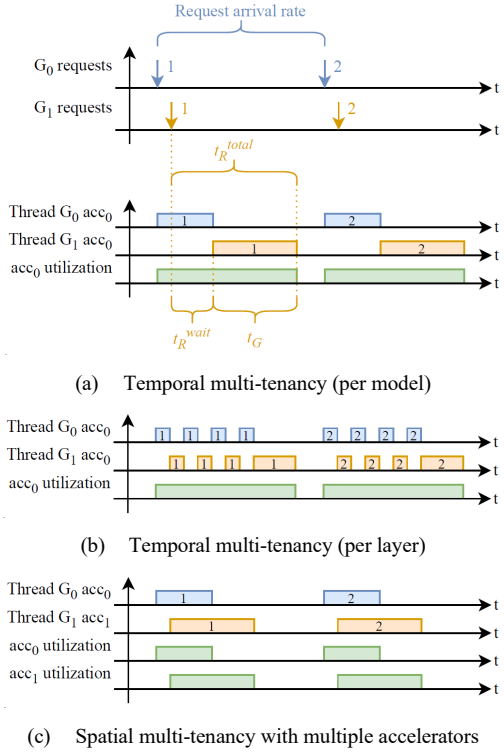


Figure 2 Different access control configurations in a multi-DNN multi-accelerator system.

can be evaluated in such a system (t_R^{wait}: request arrived but is waiting to be executed, t_G: actual model execution time, $t_R^{total} = t_R^{wait} + t_G$). For example, when only one accelerator is available, model-wise (Fig. 2a) or layer-wise blocking (Fig. 2b) modifies both t_R^{wait} and t_R^{total}. When two accelerators are available (Fig. 2c), threads assigned to different accelerators can execute the requests in parallel. The last configuration with two accelerators already shows one of the limitations of this approach, which we address in Section IV: once acc_0 finishes executing the G_0 request, it sits idle when it could be executing part of the G_1 request.

To explore the design space of possible SoC configurations, we parameterize 1) the number of accelerators M, 2) the size of the accelerators, between *Small* (16x16 PEs, 256 kB scratchpad) and *Big* (32x32 PEs, 512 kB scratchpad), and 3) the L2 cache size (1024, 2048, and 4096 kB). From the SW point of view, we also parameterize the mutex access control per accelerator (model- or layer-wise). All HW versions were implemented on a ZCU111 AMD FPGA development board, with a clock of 100 MHz[1].

For each combination of HW/SW parameters, we explore the behavior of the system under different benchmarks, each with different request arrival rates: 1) *Simple*, where one request

[1] We fixed the same frequency for all of them to focus our analysis only on the SoC-level parameters. Also, some configurations are excluded because these did not fit onto the FPGA

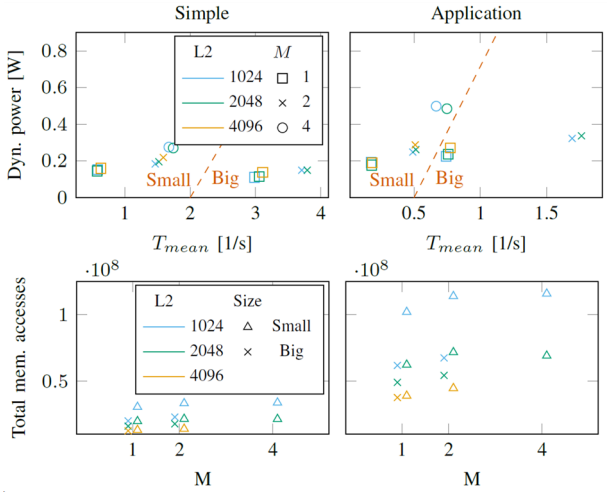


Figure 3 Measurements for different request arrival rates, grouped by L2 cache size, amount of accelerators available (*M*) and accelerator size.

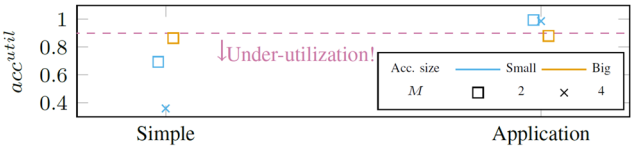


Figure 4 Mean accelerator utilization comparison.

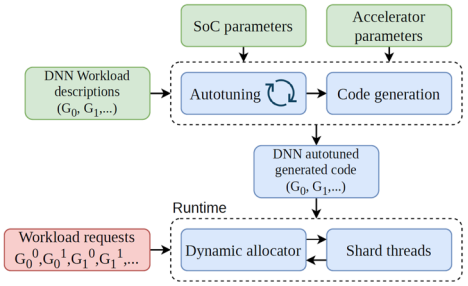


Figure 5 Proposed workflow for the autotuned-based distribution of DNNs across multiple accelerators.

is generated for each workload every second, and 2) *Application*, simulating the application's behavior, where every second one request for the YOLO model and a batch of four requests[2] for the VGG workload is generated (similar to the multi-stream arrival rate used by MLPerf Tiny [23]).

C. Measurements

We executed each benchmark for 3 seconds and then waited until all requests were processed (we call the total elapsed time $t_{benchmark}$). We also recorded the power consumption of the FPGA and the traffic between the L2 cache and the external DRAM during the entire execution. For each model's request, we record all metrics defined in Fig. 2, and define the mean throughput across all N models as Equation (1), with R_n being the amount of requests received for model *n* and $t_{R_r}^{total}$ the total time needed to finish request R_r.

$$T_{mean}[1/s] = \frac{1}{N}\sum_{n=0}^{N}\left(\frac{1}{R_n}\sum_{r=0}^{R_n}\frac{1}{t_{R_r}^{total}}\right) \quad (1)$$

Fig. 3 shows the behavior of different HW versions under different request arrival rates[3]. Increasing *M* brings a significant throughput improvement, especially during the *Application* benchmark, where several simultaneous requests can be parallelized. Similarly, increasing the size of the accelerators brings even bigger benefits: for example, having only one *Big* accelerator is already better than having 4 *Small* ones both in terms of mean throughput *and* power.

Increasing the L2 cache size improves the mean throughput, as less data needs to be exchanged with the external memory. This can also be appreciated in terms of external memory accesses (reads + writes), as these are reduced when the L2 cache is increased. But interestingly, these also increase when more accelerators are introduced. This is attributed to the shared nature of the L2 cache; concurrent execution induces cache line evictions, and data utilized by one accelerator may be displaced by the memory demands of others.

Although we do not observe a change in T_{mean} when blocking the accelerators per layer or per model, we do observe changes in t_G and t_R^{wait} [4]. The behavior is in line with the expected one presented in Fig. 2.

$$acc^{util} = \frac{1}{M}\sum_{i=0}^{M}\frac{t_{acc_i}^{busy}}{(t_{benchmark} - t_{idle})} \quad (2)$$

Finally, the mapping tool in [22] takes care of the individual PE utilization, so we analyze the mean accelerator utilization from a systems perspective (Equation (2)). $t_{acc_i}^{busy}$ represents the time accelerator *i* was busy processing a layer, and t_{idle} the time no accelerator is busy because there are no more requests to process. By subtracting t_{idle} from $t_{benchmark}$, acc^{util} only measures utilization while there are pending requests to process. Conceptually, if an accelerator is not being used while others are, the utilization for this accelerator decreases, as it could be potentially used to offload some computation from the other busy accelerators.

Fig. 4 shows that some configurations present an under-utilization of their accelerators ($acc^{util} < 0.9$). This happens in the *Simple* benchmark with both accelerator sizes, and in the *Application* benchmark with the *Big* size. In the rest of the cases, the accelerators are already constantly occupied because the arrival rate is too fast compared to the time needed by one accelerator to process one request.

IV. Proposal

Fig. 5 presents our proposal to improve the utilization of the idle resources identified in Section III-C. The compilation process receives the description of each DNN workload, the parameters of the SoC (L2 cache size, number of accelerators), and of the accelerator (for our example with Gemmini accelerators: number of PEs, size of internal scratchpad

[2] We fixed the VGG model requests to four to ensure reproducible results across different benchmark executions with different parameters and HW configurations.

[3] We only report dynamic power, as in an FPGA, the static power is fairly constant across all HW versions.

[4] Figure is omitted because of space limitations.

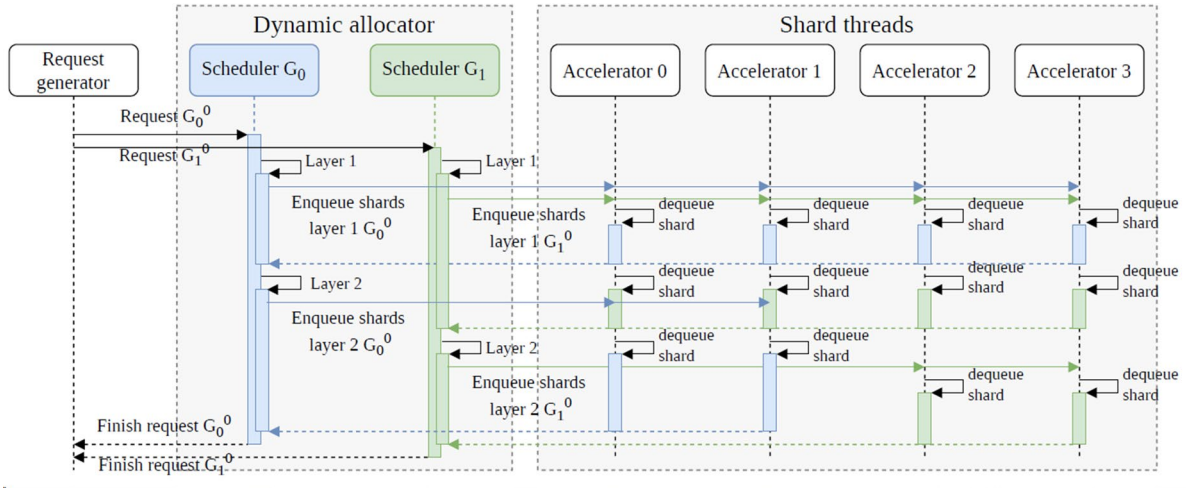
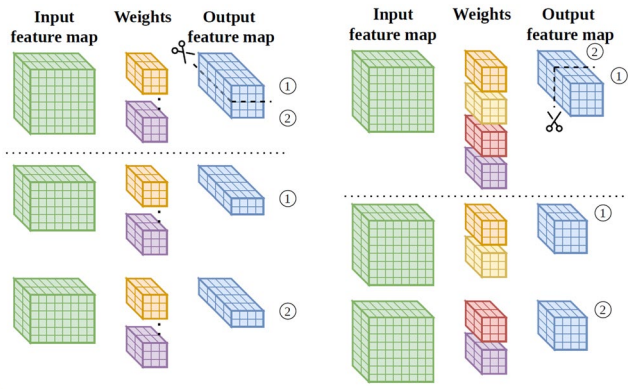


Figure 6 Example of the message passing between threads in the proposed RTOS runtime configuration.



(a) Row partition (b) Channel partition

Figure 7 Partition options examples for convolutions.

memories, etc). Then, the autotuning process finds the best way to partition each layer of the model across the available accelerators using AutoTVM [9]. During runtime, requests are received for each workload, and the dynamic allocator assigns the partitions (or *shards*) of each layer to a different accelerator.

A. Autotuning for partitioning point identification

AutoTVM has been proven to be a reliable tool for tuning the execution of tensor operations on both commercial and custom HW. This tuning process modifies the tensor operation by applying loop transformations and evaluates each candidate on the actual HW to find the one that minimizes its execution latency. Indeed, this was already applied to the Gemmini accelerator for the tuning of convolutions and other layers [24], but it only focused on one accelerator.

Although initially developed for the design exploration of loop transformations like tiling and reordering (internal to each tensor operation), we extended AutoTVM to consider two new optimization dimensions: how the Output Feature Map (OFM) of a convolutional layer is partitioned (channel- or row-wise [25]) and how many partitions are generated (limited to the maximum number of accelerators). Instead of using heuristics to decide how to partition each layer, we integrate this decision into the AutoTVM optimization framework. AutoTVM then finds the partition methodology that minimizes the latency for each layer (these measurement already encompasses the memory traffic overhead of sharing data between multiple accelerators). The one-time overhead of running this process for a particular DNN is analyzed in Section V-B.

Fig. 7 illustrates the differences between the partitioning configurations explored by our method. Using row partition (Fig. 7a) reduces the size of the input and output feature maps needed by each shard, but channel partition (Fig. 7b) can also be used to reduce the size of the weight tensors needed by each partition. The first layers from a DNN, which tend to have smaller weight tensors but larger OFMs, may benefit from row partitioning, in contrast to the last layers from the network, which may be better suited for channel partitioning.

This step only finds the best partitioning configuration for each layer. The actual shard-accelerator assignment is done at runtime, as explained in the next Section.

B. Runtime dynamic allocation of shards

To be able to assign these shards to the available accelerators dynamically during runtime, we expanded the RTOS configuration presented in Section III-B, originally thought for model parallelism. For each model, we instantiate a *scheduler* thread, which receives a request and enqueues shards into the available accelerators through the *shard* threads' queues. Fig. 6 presents a basic example of how this system would behave during runtime. Each scheduler thread first enqueues the shards of its respective first layers. The shard threads then process them until their queues are empty, and report back to the original scheduler threads every time a shard is finished. During the execution of both models' second layers, because there are two accelerators idle, the scheduler thread for

```
Require: accum_load, accum_lat, method, next_acc
 1: for layer in G_x do
 2:     assigned_acc = []
 3:     for shard in layer.shards do
 4:         ▷ Get next accelerator ◁
 5:         if method = ROUNDROBIN then
 6:             acc_idx = (next_acc + 1) % M
 7:             next_acc += 1
 8:         else if method = LOAD then
 9:             acc_idx = idxmin(accum_load)
10:             accum_load[acc_idx] += 1
11:             assigned_acc.push(acc_idx)
12:         else
13:             acc_idx = idxmin(accum_lat)
14:             lat = get_mean_lat(shard)
15:             accum_lat[acc_idx] += lat
16:             assigned_acc.push((acc_idx,lat))
17:         ▷ Add shard to specific accelerator queue ◁
18:         enqueue(acc_idx, shard)
19:     ▷ Wait for all shards to finish ◁
20:     wait_for_shards_ready()
21:     ▷ Reset accelerator usage ◁
22:     if method = LOAD then
23:         for acc_idx in assigned_acc do
24:             accum_load[acc_idx] -= 1
25:     else if method = MEAN then
26:         for (acc_idx, lat) in assigned_acc do
27:             accum_lat[acc_idx] -= lat
28: return
```

Algorithm 1 Scheduler thread pseudocode.

G_1 can decide to execute the shards of its second layer on them, thus improving utilization.

Algorithm 1 presents the pseudocode for each scheduler thread. Variables *accum_load*, *accum_lat*, and *next_acc* are shared across all scheduler threads, and their access is adequately protected to avoid concurrent access. Variable *method* selects how to obtain the accelerator to which the next shard will be assigned.

The dynamic allocation comes into play during the selection of *where* to enqueue the shards of a particular layer (Line 4). A simple load balancing could be achieved by assigning shards to accelerators in a round-robin fashion (Line 6). But this method would neglect how many shards are still pending execution on each shard thread, and therefore, how many shards need to be executed on the selected accelerator before the current shard is executed.

A second approach is to assign the shards based on the already available load of each shard thread queue, and by assigning each shard to the queue with fewer pending shards (Line 9). List *accum_load* keeps track of the assigned shards per accelerator, and then resets it accordingly when the shards finish (Line 24). But this method ignores the latency of each pending shard in the queue, which is necessary to know which shard thread will become idle first.

To solve this, the assignment should then take into account the load of each queue, but in terms of *expected latency* (Line 13). The dynamic allocator can then estimate which threads will finish their pending shards faster and assign new shards to them to distribute the load.

But calculating this *expected latency* is not trivial, as variations exist in layer execution time when more than one accelerator is available in the system. Fig. 8 presents the latency of individual layers obtained from the baseline measurements from Section III-C[5]. The more accelerators there are available in the system, the more the latency of a particular layer varies across different requests. For systems with 2 and 4 available accelerators, the latency of a layer cannot be represented by a single value anymore, but by a *distribution* of possible latencies. This behavior is a result of the sharing of the memory subsystem across all accelerators, as described in Section III-C, and is in line with the observed increase in memory accesses when *M* increases shown in Fig. 3.

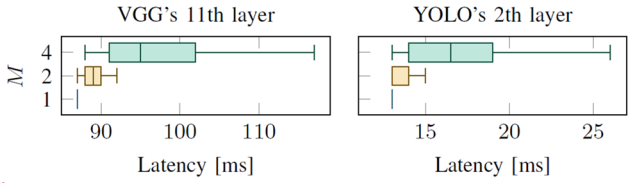


Figure 8 Distribution of latencies measured on a SoC with *Small* accelerator/s and 2048 kB L2 cache size..

To keep track of how much time each shard thread will still be busy during runtime, we need to estimate each shard's latency *every time* one is allocated. Based on the analysis of Fig. 8, we propose to use the *Mean* of previous measurements for this (Line 14}). List *accum_lat* keeps track of this *expected latency* per accelerator, and then resets it accordingly when the shards finish (Line 27).

The decision of where to assign each shard needs to be simple, as this needs to be executed by each scheduler thread every time a shard needs to be executed, and thus cannot insert unnecessary overhead in the execution of each request. The *Round-robin* allocation method is the simplest, as it only requires returning the next accelerator. The other two methods (*Load* and *Mean*), although comparing different metrics, are also simple, as these only require *M*-1 lookups to find the accelerator with the least pending latency or the least queued shards. This makes these three methods lightweight and suitable for implementation on resource-constrained devices.

V. Evaluation

In this section, we evaluate the impact of our proposal by applying our method to the VGG model of our representative use case. We only use our method on the VGG model because in a real scenario this model would be executed *for each* detected object in the scene, so it would benefit more from a faster completion time thanks to distributed execution across multiple accelerators. We only focus on hardware setups where the utilization is lower than 0.9 (Fig. 4), as these are the cases that can be improved by taking advantage of idle accelerators. The model parallelism measurements presented in Section III are used as baselines.

A. Throughput and utilization improvements

Fig. 9 shows, for each HW version and benchmark, how much the mean accelerator utilization and throughput are

[5] We observe the same behavior in all layers and across all HW versions. On average, because of competing access to memory, using two and four accelerators increases the mean measured latency for each layer by 31% and 91% respectively.

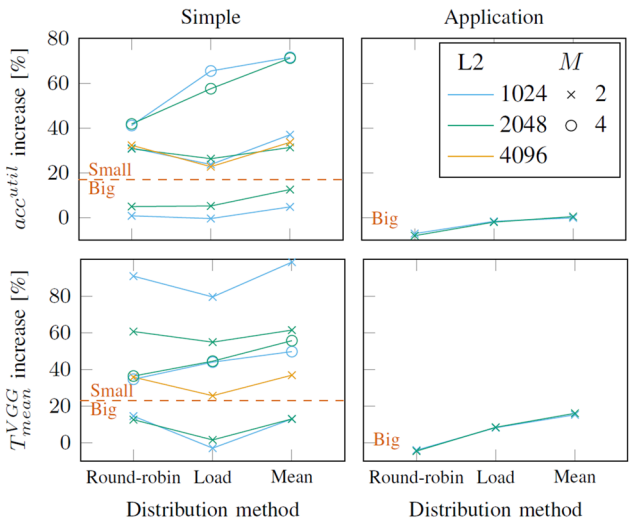


Figure 9 Improvements in acc^{util} and throughput for the VGG model using different distribution strategies for both benchmarks (compared against the corresponding best result from Section III).

improved when compared against the baseline from Section III-C. The dynamic scheduler using the *Mean* latency estimation for the allocation of shards is constantly better than both the *Round-robin* distribution of shards and the one based only on the *Load* of each queue. The improvement increases as more accelerators are available, which makes sense, as these are the cases with more idle accelerators, which provide more opportunities for improvement. For the *Simple* benchmark, our dynamic allocator using the *Mean* method is able to achieve a mean 37 % increase in acc^{util}. For the *Application* benchmark, the increase is only 0.2 %, as these HW versions already had a high accelerator utilization (Fig. 4). Overall, our dynamic allocator is able to achieve a 29 % mean increase in acc^{util} when compared against the model parallelism baseline, and a 10 % increase when compared against the *Round-robin* method.

In terms of throughput, our dynamic allocator method using the *Mean* method achieves a mean 40 % increase when compared against the model parallelism baseline, and a 9 % improvement against the *Round-robin* distribution method.

For both accelerator utilization and throughput, the *Load* method is not reliable, as only counting the number of shards assigned to each accelerator is not enough to adequately decide which accelerator will become idle faster. Our *Mean* method always surpasses it.

Improvements are significant for the SoCs with smaller accelerators, but lower for the other HW versions. A significant reason for the underutilization of the *Big* accelerators comes from the more similar execution time of both models, which results in little time where one accelerator is idle while the others are still busy. This can be seen more clearly in Fig. 10. On the left, during the first request, accelerators 0 and 2 are idle during the entire duration of the request, and there is a significant amount of time accelerator 3 is still executing the VGG model (orange) while accelerator 1 has already finished

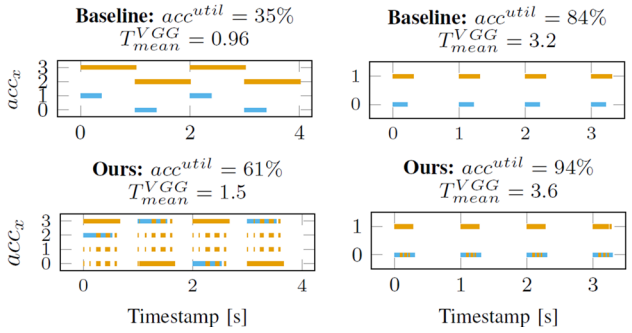


Figure 10 Execution traces for the Simple benchmark for an SoC with 4 Small accelerators (left) and one with 2 Big accelerators (right), showcasing the increase in acc^{util} and T^{VGG}_{mean} by using our method.

the execution of the YOLO model (blue). On the right, because the VGG model is executed faster (as this model is mapped more efficiently onto the *Big* accelerators), there is less time one accelerator is still busy, and the other one idle (as defined in Equation (2), the time where no request is available is not taken into account as idle time for the utilization calculation). The impact of our work can be seen more clearly in the case with 4 *Small* accelerators, but it is still able to increase both utilization and throughput for the case with 2 *Big* accelerators.

B. *Autotuning overhead*

Executing autotuning processes like AutoTVM introduces a one-time compilation overhead because the tuning process evaluates each possible candidate on the HW to find the one that minimizes the latency. Given our new autotuning parameters defined in Section IV-A (type of partition, row or channel, and number of partitions, limited to *M*), the maximum number of possible partitioning candidates that need to be measured on the HW can be calculated. For a given DNN workload *G* containing *L* layers, for an SoC with *M* accelerators, the maximum number of candidates to measure is defined by Equation (3).

$$candidates = L \times (2 \times M - 1) \quad (3)$$

For the VGG model (*L*=16, only convolutions) on a SoC with *M*=4, this yields 112 possible candidates. In our experiments running AutoTVM with our modifications on a 12-core i7 CPU (3.20GHz) server with 32 GB of RAM, the entire process required approximately 45 minutes. Given the performance improvements demonstrated in Section V-A, this one-time cost is well justified.

VI. Conclusion

This paper first explored the HW design space for a multi-DNN multi-accelerator SoC system and proposed an RTOS configuration to achieve automatic model parallelism. From the FPGA implementation results, we extracted insights and identified that model parallelism underutilized the available accelerators. In order to improve this, we proposed a joint compilation and runtime methodology to first find the optimal

partitioning points for each tensor operation during compile time using autotuning, and then dynamically distribute these partitions across the available accelerators using a lightweight scheduler. We explored different decision techniques for the proposed dynamic allocator scheduler, and demonstrated that we can achieve a 29 % and 40 % improvement in mean accelerator utilization and throughput, respectively, over the model parallelism baseline. When compared against a round-robin distribution of partitions, our dynamic allocator is able to achieve a 10 % and 9 % increase in mean utilization and throughput, respectively.

Although this work focused mostly on the object detection aspect of an autonomous driving application, a future extension could evaluate the proposed methodology on a full-scale autonomous driving application, where additional DNN workloads need to be orchestrated for other tasks like object tracking and trajectory prediction. Additionally, only the mean of the distribution of measured latencies was used to guide the runtime allocation of shards. Other metrics of the distribution (e.g., different quantiles) may present different trade-offs and are worth analyzing.

REFERENCES

[1] A. Mousavian, D. Anguelov, J. Flynn, and J. Kosecka, "3d bounding box estimation using deep learning and geometry," CoRR, vol. abs/1612.00496, 2016. [Online]. Available: http://arxiv.org/abs/1612.00496

[2] C.-J. Wu, D. Brooks, K. Chen, D. Chen et al., "Machine learning at facebook: Understanding inference at the edge," in 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2019, pp. 331–344.

[3] C. Wang, Y. Bai, and D. Sun, "CD-MSA: Cooperative and Deadline-Aware Scheduling for Efficient Multi-Tenancy on DNN Accelerators,"IEEE Transactions on Parallel and Distributed Systems, vol. 34, no. 7,pp. 2091–2106, Jul. 2023.

[4] C. Gao, Y. Wang, C. Liu, M. Wang et al., "Layer-Puzzle: Allocating and Scheduling Multi-task on Multi-core NPUs by Using Layer Heterogeneity," in 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). Antwerp, Belgium: IEEE, Apr. 2023, pp. 1–6.

[5] Z. Zhao, N. Ling, N. Guan, and G. Xing, "Miriam: Exploiting elastic kernels for real-time multi-dnn inference on edge gpu," in Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems, ser. SenSys '23. New York, NY, USA: Association for Computing Machinery, 2024, p. 97–110. [Online]. Available: https://doi.org/10.1145/3625687.3625789

[6] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb et al., "Chipyard: Integrated design, simulation, and implementation framework for custom socs," IEEE Micro, vol. 40, no. 4, pp. 10–21, 2020.

[7] H. Genc, S. Kim, A. Amid, A. Haj-Ali et al., "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in Proceedings of the 58th Annual Design Automation Conference (DAC), 2021.

[8] Linux foundation. zephyr [Online]. Available: https://www.zephyrproject.org/[Accessed: 16 April 2025].

[9] T. Chen, L. Zheng, E. Q. Yan, Z. Jiang et al., "Learning to optimize tensor programs," CoRR, vol. abs/1805.08166, 2018. [Online]. Available: http://arxiv.org/abs/1805.08166

[10] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture, ser. ISCA '20. Virtual Event: IEEE Press, Sep. 2020, pp. 940–953.

[11] Y. H. Oh, S. Kim, Y. Jin, S. Son et al., "Layerweaver: Maximizing Resource Utilization of Neural Processing Units via Layer-Wise Scheduling," in 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Feb. 2021, pp. 584–597.

[12] Y. Choi and M. Rhu, "PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units," in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). San Diego, CA, USA: IEEE, Feb. 2020, pp. 220–233.

[13] S. Ghodrati, B. H. Ahn, J. Kyung Kim, S. Kinzer et al., "Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks," in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Oct. 2020, pp. 681–697.

[14] H. Kwon, L. Lai, M. Pellauer, T. Krishna et al., "Heterogeneous Dataflow Accelerators for Multi-DNN Workloads," in 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Feb. 2021, pp. 71–83.

[15] S.-C. Kao and T. Krishna, "MAGMA: An Optimization Framework for Mapping Multiple DNNs on Multiple Accelerator Cores," in 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). Seoul, Korea, Republic of: IEEE, Apr. 2022, pp. 814–830.

[16] X. Zhang, C. Hao, P. Zhou, A. Jones, and J. Hu, "H2H: Heterogeneous Model to Heterogeneous System Mapping with Computation and Communication Awareness," Apr. 2022.

[17] S. Karl, A. Symons, N. Fasfous, and M. Verhelst, "Genetic Algorithm-based Framework for Layer-Fused Scheduling of Multiple DNNs on Multi-core Systems," in 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). Antwerp, Belgium: IEEE, Apr. 2023, pp. 1–6.

[18] S. Zheng, S. Chen, and Y. Liang, "Memory and Computation Coordinated Mapping of DNNs onto Complex Heterogeneous SoC," in 2023 60th ACM/IEEE Design Automation Conference (DAC). San Francisco, CA, USA: IEEE, Jul. 2023, pp. 1–6.

[19] J. Yang, H. Zheng, and A. Louri, "Versa-DNN: A Versatile Architecture Enabling High-Performance and Energy-Efficient Multi-DNN Acceleration," IEEE Transactions on Parallel and Distributed Systems, vol. 35, no. 2, pp. 349–361, Feb. 2024.

[20] M. Amine Hamdi, F. Daghero, G. Maria Sarda, J. Van Delm et al., "Match: Model-aware tvm-based compilation for heterogeneous edge devices," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 44, no. 10, pp. 3844–3857, 2025.

[21] M. Kirschner, F. Peccia, F. Th¨ommes, V. P. Betancourt et al., "Characterization of execution time variability in fpga-based ai-accelerators," in 2023 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), 2023, pp. 1–8.

[22] F. N. Peccia and O. Bringmann, "Integration of a systolic array based hardware accelerator into a dnn operator auto-tuning framework," in Proceedings of the 2023 Workshop on Compilers, Deployment, and Tooling for Edge AI, ser. CODAI '23. New York, NY, USA: Association for Computing Machinery, 2024, p. 21–26. [Online]. Available: https://doi.org/10.1145/3615338.3618130

[23] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman et al., "Mlperf tiny benchmark," Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks, 2021.

[24] F. N. Peccia, S. Pavlitska, T. Fleck, and O. Bringmann, "Efficient edge AI: Deploying convolutional neural networks on fpga with the gemmini accelerator," in 2024 27th Euromicro Conference on Digital System Design (DSD), 2024, pp. 418–426.

[25] E. Baccour, N. Mhaisen, A. A. Abdellatif, A. Erbad et al., "Pervasive AI for IoT applications: A survey on resource-efficient distributed artificial intelligence," IEEE Communications Surveys & Tutorials, vol. 24, no. 4, pp. 2366–2418, 2022.

Clock-glitch Fault Characterization by Timing Comparison with Laser Fault Injection

Ludovic Claudepierre, Edna Rocio Ferrucho-Alvarez, Laurent Le Brizoual, Laurent Pichon
Univ Rennes, CNRS, IETR (Institut d'Electronique et des Technologies du numéRique)
UMR 6164, Rennes, France
ludovic.claudepierre.1@univ-rennes.fr

***Abstract*—Since clock-glitching and the voltage-glitching are non-localized techniques of fault injection, it is difficult to accurately characterize their fault effects. In contrast, laser fault injection is a localized technique in which the targeted element of the microprocessor is known. In this article, we present a method that exploits the advantages of laser fault injection to characterize the glitch fault effects. We first make an hypothesis about the physical location of the sensitive part that induces the fault by glitch injection. Then we compare the fault timing and execution timing for glitch and laser platforms. If the delay between the fault and the execution is the same for both platforms, the initial hypothesis is supported.**

Keywords-Fault attack, Laser fault injection, Clock glitch, Fault characterization.

I. INTRODUCTION

Fault injection is defined as the introduction of a physical disturbance at microelectronic level with the objective of inducing a malfunction at physical level called a fault. This fault propagates through logic and microarchitecture to finally reach software level to provoke alteration on the program executed by the microprocessor. Fault injection can be used to analyze the robustness of electronic circuits and embedded systems or to exploit vulnerabilities. Faults have the capacity to corrupt data or instructions which can weaken security mechanisms or leads the program to crash. There are several techniques to physically disrupt a microprocessor, including voltage glitch [1], clock glitch [2] [3], electromagnetic fault injection [4] [5], and laser fault injection [6]. Voltage and clock glitches are global injection methods as the perturbation propagates though the whole microprocessor (power line or clock tree). The two methods involve the injection of a brief disruption, either into the voltage line or the clock signal. This disruption propagates through the circuit until it reaches a critical element whose failure results in a fault. Electromagnetic fault injection is a localized method in which the microprocessor is exposed to a local short-range electromagnetic pulse inducing a fault. Laser fault injection involves pointing a focused laser beam at a designated area of the circuit. This causes a transient photocurrent responsible for the switching of a transistor leading to a disruption at logic level.

Manuscript received May 08, 2026; revised July 14, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.135

Fault effects can be observed at different levels of abstraction. The characterization step consists in understanding and modeling the impact of fault injection at a designated abstraction level. Frequently, the observation is only made on the registers at ISA (Instruction Set Architecture) level. Consequently, the fault model is only based on the comparison of the registers between nominal execution and execution with fault. This step facilitates the elaboration of cyberattacks potentially complex and hard to prevent. However, performing deeper analysis is really challenging. In particular, the propagation mechanism from the physical level to the software level is hard to explain with certainty. This step is even more challenging for glitch attack as the precise area which is failing is unknown. The method presented in this paper aims to address this challenge by using the advantage of the localized laser injection to better describe the fault glitch mechanism.

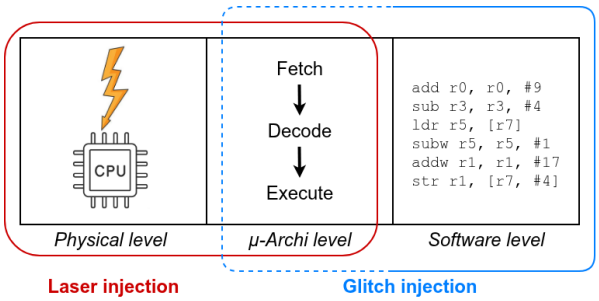


Figure 1: Layer information according to the injection techniques.

When performing glitch fault injection, an hypothesis at the microarchitectural level can be proposed to explain specific fault effects at ISA level. (Figure 1). In particular, an assumption can be made about the pipeline stage sensitive to the glitch injection. Accordingly, a list of the physical elements involved in that stage that could be critical can be made (flash memory, RAM memory, registers). Then, the validity of this hypothesis can be verified by using laser injection. Indeed, as a located fault injection technique, the fault effects observed at ISA level are directly associated with a specific physical element and consequently with certain pipeline stages. Since glitch and laser injection do not result in the same fault effect, the comparison is made by analyzing the fault timing. The new

method developed to perform such analysis is called GARLIC (Glitch chARacterization by Laser Injection Comparison).

In section II, related works on characterization methods and also on clock-glitch capabilities are presented. Then, every step of the GARLIC method is detailed in section III. In section IV and V, the glitch platform TRAITOR and the laser platform are presented, along with the characterization of the fault models chosen to apply GARLIC. In section VI, the conditions of the application of our method to characterize the skip-repeat fault of TRAITOR are detailed. Finally, in section VII, results and conclusions of this study are exposed.

II. Related Work

A. Characterization

In order to construct a fault attack on instruction flow or data flow, it is necessary to characterize the fault effect so as to understand the mechanism at a given level of abstraction. Furthermore, in some cases, we can get information about the propagation mechanism through the different abstraction levels that finally induces a faulty behavior at software level. The precise architecture of the targeted microcontroller is not accessible for devices on-the-shelf. Therefore, the utilization of a simplified model is the only way to depict the fault. The characterization of the observed effects can be made at physical level [7], RTL [8] [9] and ISA level [10].

When characterizing a fault effect, the selection of the test code is a crucial step. The instructions of the test code have to be chosen according to the expected effect - whether it is a skip, skip-repeat, or instruction corruption. Alle Monne *et al.* [11] propose an automated methodology for synthesizing characterization programs using fault model checking on CPU RTL design. Trouchkin *et al.* [12] propose a methodology for determining, in the case of instruction corruption, which part of the instruction is corrupted and in which stage of the pipeline the fault occurs. Moro *et al.* [13] compare the results of fault injection campaigns with simulation where instructions are replaced by alternative ones to determine if there is a match at ISA level. Instrumentation of the program can facilitate the characterization process, for example by enabling automatic identification of the most common fault models [14].

B. Clock-glitch fault injection

Tools such as the ChipWhisperer [15] inject glitches that can provoke faults that are generally described as a consequence of a timing violation or a setup violation. The effects can include instruction skip [16], skip and repeat or instruction alteration as demonstrated by Alshaer *et al.* [17]. Claudepierre *et al.* [3] proposed the platform TRAITOR using a different way to perform a clock glitch by interrupting a rising edge of a clock signal. Regarding this type of glitch, Marotta *et al.* [18] explain that for flip-flops to correctly sample incoming data, the clock signal must reach a given threshold; otherwise, a fault occurs.

III. Method

The objective of the GARLIC method is to obtain information at physical level regarding the fault effects observed when performing fault injection with non-localized injection techniques. This method is applied to a specific fault effect observed on a specific platform. For the experiments a GPIO is used as a trigger to synchronize the injection platform and the target. This signal is the reference for timing measurements and every fault timing is expressed as the delay between this trigger and the fault injection. In order to know the instruction execution duration *i.e* the duration that the instruction remains in the execute pipeline stage, we can either look at the instruction set documentation or make a measurement for example by using the DWT (Data WatchPoint and Trace Unit) debug register if the microprocessor is ARM. Making a measurement is the more accurate option because it takes into account potential stalls, unlike the documentation data which only provides theoretical values. First, we establish a list of terms used throughout this article. We only consider cases where instructions are aligned. Consequently, a word is defined as a group of complete instructions.

- Execution duration of an instruction: The number of cycles that the instruction remains in the execute stage of the pipeline, as described in the documentation.
- Execution duration of one word: The sum of the execution duration of the instructions contained in the word.
- Execution timing of a targeted code: The list of the execution duration for the instructions contained in this code.
- Fault delay: Delay between the trigger and the fault. Depending on the injection type, it can be a number of clock cycles or a delay in nanoseconds.
- Fault timing: List of the fault delays for the instructions composing the test code.

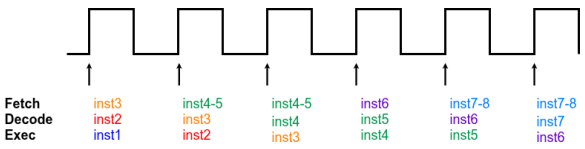


Figure 2: Example of instructions progression in a 3 stage pipeline.

A. Method overview

Laser fault injection is characterized by its spatial locality. The targeted element is used in some specific pipeline stages, for example flash memory is only used during the prefetch. Therefore, faults occur at specific pipeline stages or during the transfer between two stages. Figure 2 illustrates the example of the progression of instructions in the 3 stage pipeline of a Cortex-M3 microprocessor with a 32 bit prefetch buffer and using Thumb-2 instruction set allowing 16 bits instructions. A different color is attributed to each word, thus two instructions with the same color are stored in the same word (for example instructions 4 and 5). An instruction remains in the execute stage during a given number of clock cycles. This imposes on the following instruction to remain in the decode stage for the

same amount of time. Consequently, it also imposes the number of clock cycles that the next instructions remain in the fetch stage. Considering this, the delay between faults on two successive instructions at a given stage depends on the execution duration of the instruction presently in the execute stage. Therefore, by comparing the fault delays for every instruction in the test code with the execution timing, we should be able to determine which instruction is in the execute stage when a fault occurs.

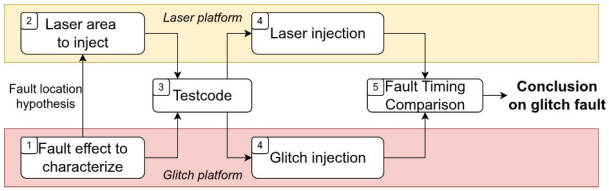


Figure 3: GARLIC method overview.

As illustrated in Figure 3, the method consists of 5 steps. First, the glitch platform and the fault effect we want to characterize are selected. A hypothesis on the potential mechanism responsible for this effect is formulated. From this, the potential fault location is deduced. Second, the location for laser injection is chosen based on this assumption. Third, the test code is written taking into account the constraints imposed by the glitch fault effect to analyze and the laser injection area. Then, the injection campaigns on both platforms are performed. Finally, the fault timing of both injection campaigns are compared. By examining the differences and the similarities, conclusions about the fault mechanism can be drawn, leading either to the support or the rejection of the initial hypothesis.

B. *Glitch platform and type of fault*

The first step is to select a non-localized injection platform to work with. It could be a clock glitch or a voltage glitch platform. The type of fault to be analyzed must be repeatable with a given set of parameters and observable on multiple instructions with different execution duration. At this stage, a hypothesis is formulated concerning the pipeline stage disrupted by the glitch injection.

C. *Area for Laser fault injection*

The hypothesis on the pipeline stage affected by the glitch fault guides the choice of the laser spot location. Therefore, we have to select a physical area whose sensitivity is associated with a specific element (flash memory, RAM, registers). In addition, this element has to be used in a single stage of the pipeline to easily be linked to the fault timing and the involved pipeline stage.

D. *Test code*

The elaboration of the test code for the method application is an essential part of the process. This test code has to be designed to ensure the observation of faults effects with both glitch and laser platforms. Furthermore, the method relies on the comparison between the execution duration of instructions and the delay between the fault of two successive instructions. To obtain a pattern, the test code has to contain successive instructions with different execution duration. As shown in Figure 4, this pattern is observable for execution timing and fault timing with a delay between them. For example, there is one cycle between the moment when instruction 3 is faulted and the moment instruction 4 is faulted. Thus, we know that the instruction executed during that time stays only 1 cycle in the execute stage. By making this comparison for each faulted instruction, we can make the connection between execution duration and fault delay. The execution pattern has to be non-periodic to ensure a single match between execution and fault timing. We note that the timing can be observed for instructions or words, depending on the fault effect on which `GARLIC` is applied.

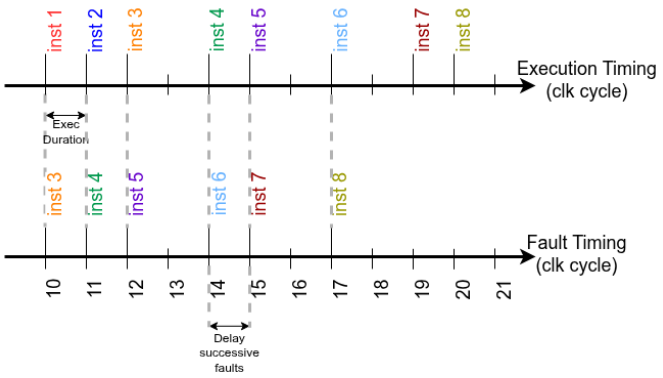


Figure 4: Principle of comparison between fault timing and execution timing.

E. *Injection Campaign and results*

Prior to the method application, a test on the glitch injection platform is performed. The objective of this experiment is to determine the parameter values necessary to achieve the fault effect under study. Also, laser injection parameters are defined to make timing located faults with a precision inferior to one clock cycle. After these steps, the fault injection campaigns can be performed on the test code for both platforms. The campaign sweeps the injection delay such that every instruction is faulted.

First, for laser and glitch injection, the delay between the fault and the execution is measured for every instruction. Then, we can determine which instruction is in the execute stage when faults are observed. This can give an indication of the pipeline stage targeted by the fault. Finally, the fault timing for the laser and the glitch injection are compared with an analysis of similarities and differences between them.

IV. CLOCK-GLITCH PLATFORM: `TRAITOR`

In this part the clock-glitch injection platform `TRAITOR` is presented and the fault effects observable when attacking simple codes are detailed.

A. *Hardware setup*

`TRAITOR` is a multi clock-glitch platform implemented on an Artix-7 FPGA (Figure 5). The output signal replaces the clock source of the targeted microcontroller [3]. Also, as PLL

is a countermeasure to clock glitches, it has to be deactivated. In this example and for the rest of this article, the device under test (DUT) is a STM32 Discovery board with a STM32F100RB Cortex-M3 processor. The clock is provided by a 8 MHz quartz crystal corresponding to a 125 ns clock period.

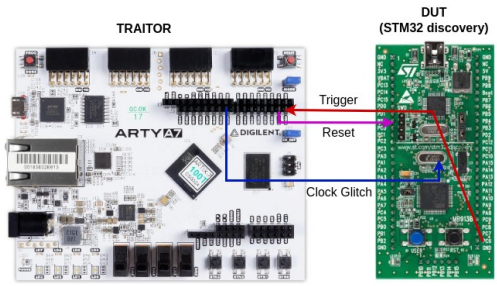


Figure 5: TRAITOR platform and connection to the DUT.

The glitch of `TRAITOR` is synchronous and can be described as an interrupted rising edge. Each glitch is defined by its amplitude, which is the value image of the physical voltage of the glitch (from 0 V to Vcc), and the delay when the glitch is injected (Figure 6). This glitch results of a combination of logic operations on two delayed clock signals.

In the context of multi-glitch attacks, the number of successive glitches is an additional parameter to consider. In that case, the attacker programs `TRAITOR` with the amplitude (identical for all the glitches of the attack), the delay and the size of each group of successive glitches. The most commonly observed effect when attacking with `TRAITOR` [3] is the skip of all the instructions contained in one word and the repeat of all the instructions contained in the previous word. As a matter of concision we will talk about skipped word and repeated word, respectively. Consequently, the following observations have been made regarding words and not instructions alone.

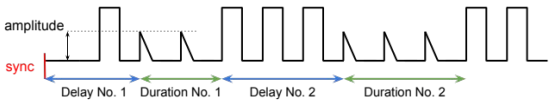


Figure 6: TRAITOR output signal characterized by the glitch amplitude, the delay of injection for each glitch burst and its duration.

B. Fault effect observed

First, the various fault effects obtained by injecting faults with `TRAITOR` have to be analyzed. A fault campaign is launched on two simple codes compiled into the Thumb-2 instruction set only containing arithmetic instructions: one code with only 16 bits instructions (Listing 1) and the other one with only 32 bits instructions (Listing 2). The effects of the amplitude over the fault effect is investigated for each delay between the first and the last faulted instruction. For these two codes, we chose aligned instructions to avoid any hybridization of instructions as observed by Alshaer *et al.* [17]. Indeed, this effect makes the identification of fault models more complicated and has no interest in our study. We also chose non-idempotent instructions to make possible the observation of potential repeated instructions when faulting. For this experiment, faults are injected at every clock cycle such as each word is faulted. The range of glitch amplitude values is from 420 to 530.

```
...
nops
nops
adds r0, r0, #11
subs r3, r3, #4
adds r1, r1, #13
subs r4, r4, #5
...
nops
nops
...
```

Listing 2: Code with 16 bits instructions.

```
...
nopw
addw r0, r0, #11
subw r3, r3, #4
addw r1, r1, #13
subw r4, r4, #5
...
nopw
...
```

Listing 1: Code with 32 bits instructions.

The results of the injection campaign with `TRAITOR` are plotted in Figure 7. The different types of fault are represented by different colors, with respect to the delay and amplitude values. First, we observe that regardless of the instruction size, the fault effect is most of the time a skip with some repeat side effects. The *n* value is related to the index of the skipped word.

In the case of 16 bits instructions (Figure 7b), the only observable fault model is the $skip_n$ /$repeat_{n-1}$ (red). As the words contain two instructions of 16 bits, each one executed in 1 cycle, this fault can only occur every 2 cycles. This type of fault is observable for amplitude values from 440 to 500.

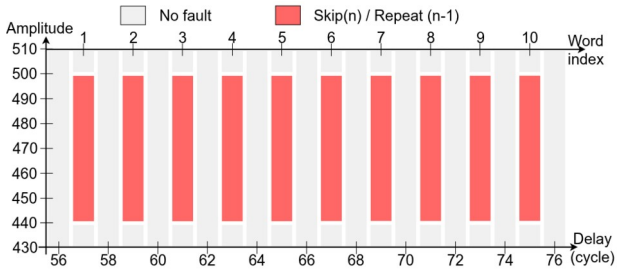
In the case of 32 bits instructions (Figure 7a), fault can be observed at every cycle. It makes sense as each word contains only 1 instruction of 32 bits that is executed in 1 cycle. The $skip_n$/$repeat_{n-1}$ (red) is also observable every cycle but in a smaller amplitude range of 440 to 455. Every 2 cycles, $skip_n$/$repeat_{n-2}$ (blue) appears for amplitudes of 455 to 465. A $skip_n$/$repeat_{n-4}$ type of fault is also observable every 4 cycles for higher amplitudes. Other unidentified effects are detected for higher amplitudes as well.

Among the various fault effects, the only one that is observable regardless the instruction size is the $skip_n$ /$repeat_{n-1}$. For the application of `GARLIC` method, it is necessary to write a test code containing words with variable execution duration. We made the choice to only characterize glitches on addition and subtraction. Consequently, the test code must contain a mix of instructions with different sizes to obtain words with variable execution duration. For that reason, we chose to study the $skip_n$ /$repeat_{n-1}$ fault. In the remainder of this article, we will refer to that type of fault as *skip/repeat*.

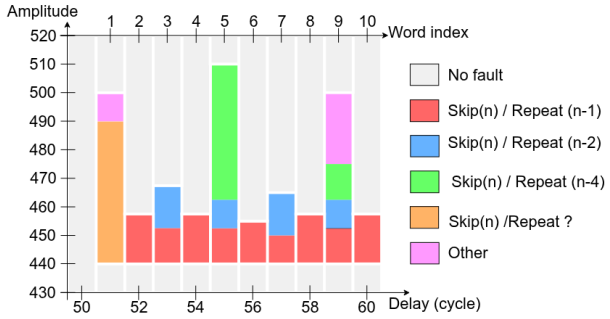
C. Hypothesis on the fault mechanism

Before going further, it is necessary to make an assumption about the physical location of the fault induced by the clock glitch on the selected target. First, the skip/repeat appears on a whole word and not on separate instructions. Thus, the possibility of faulting the decode and execution stages can be excluded. Two transfer steps could be disrupted: the transfer

from the flash memory to the prefetch buffer and the transfer from the the prefetch buffer to the fetch stage. The location of the flash memory on our microcontroller is easily identifiable. Consequently, the fetch of instructions from the flash memory to the prefetch buffer can easily be faulted by laser injection. For these reasons, we chose to investigate on the hypothesis that `TRAITOR` glitch disrupts the transfer from the flash memory to the prefetch buffer.



a. Faults on 16 bits arithmetic instructions



b. Faults on 32 bits arithmetic instructions

Figure 7: Type of faults for 16 and 32 bits arithmetic instructions with n the word index.

V. Laser Fault Injection

A. Laser bench and target

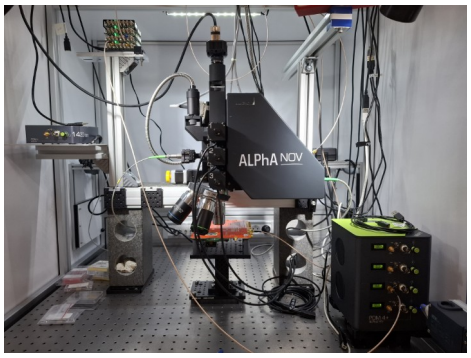


Figure 8: Laser fault injection bench.

The laser bench used in our experiments (Figure 8) has a 1064 nm wavelength source that is used with a 5x objective. This optical part is mounted on a three-axis motorized turntable. The maximum power output is 5 W. The injection delay and the pulse width can be adjusted with a precision of 0.1 ns.

The targeted microcontroller is the same as that used in the clock-glitch attack (STM32F100RB Cortex-M3). The processor is interfaced on a Donjon Scaffold board. The clock source is an 8 MHz quartz crystal which corresponds to a 125 ns clock period. Figure 9 shows an infrared image of the target after mechanical decapsulation. The main parts of the microcontroller have been identified: the analog part (in blue) with the power and clock management; the flash memory (red) where the instructions of the targeted code are stored; the logic part (yellow) where the instructions are launched from flash memory, treated, and executed; the RAM memory (green) where data are temporarily stored during the code execution.

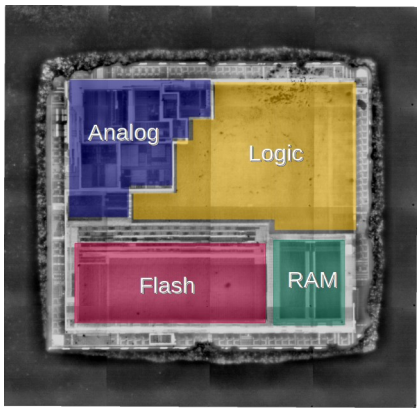


Figure 9: Infrared picture of the STM32F100 microcontroller and identification of areas

B. Parameters calibration

For the experiment, faults are injected into the flash memory of the microcontroller. Colombier *et al.* showed that this type of experiment resulted in a bit-set over the instruction opcode [19]. Since the objective of the study is a timing analysis, we have to find the best parameters to obtain temporally localized faults with a good success rate. The parameters have to be set such that one pulse can only fault one word.

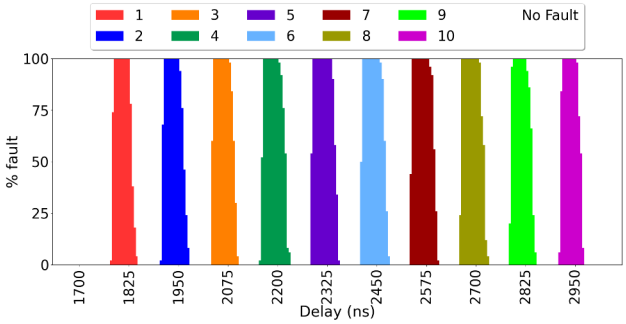


Figure 10: Percentage of faults with respect to the injection delay with power of 100 mW and pulse width of 285 ns. The colors index is related to the faulted instruction.

These conditions are met with a pulse width of 285 ns and a power of 100 mW. We tested these parameters on the piece of code used in TRAITOR characterization (Listing 2) constituted of 32 bits ADDW and SUBW instructions. The laser injection is performed on the 24th bit of the word. The fault is a bit-set

on the immediate number that consequently adds or subtracts 128 (0x80) to the register involved in the addition or subtraction, respectively. The percentage of faults with respect to the injection delay is plotted in Figure 10. We observe that the delay range for which a fault is possible over a word is around 70 ns which is less than the clock period (only 0.56 period). Therefore, the timing precision is sufficient to discriminate the faults injected on two distinct clock rising edges.

VI. Application

The GARLIC methodology is applied to determine the location of the disrupted stage when injecting fault with TRAITOR and observing a skip/repeat. This section details the parameters used for the experiment.

A. Test code

For the application of GARLIC, a sequence of instruction words with alternating execution time and with no periodicity is necessary. The instructions have also to be selected so that faults by glitch injection and laser injection are observable. Considering the arithmetic instructions used for the characterization step, we chose to write the test code with a mix of 32 bits and 16 bits addition and subtraction instructions to obtain words executed in 1 and 2 cycles, respectively. These non-idempotent instructions make the skip/repeat fault effect observable

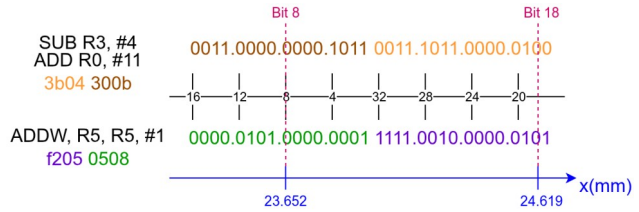


Figure 11: Localization of the targeted bits and instruction organization in the flash memory.

Concerning the laser fault injection, three constraints must be considered. First, the same bit has to be targeted all along the experiment. This means that initially, this bit has to be at '0' on every word. Second, since some of the words contain two instructions, two experiments on two different bits have to be performed: one in the 16 low-bits and the other one in the 16 high-bits. Finally, in order to make possible the identification of the fault, the fault injection must not result in a crash. The instructions are stored into the flash memory such as depicted in Figure 11. Consequently, the test code has been written so that bits 8 and 18 are set to '0' for each word. Faulting the 8th bit adds 128 to the immediate value of the instructions. Faulting the 18th bit adds 2 to the immediate value for the 16 bits instructions or add 2 on the register index Rn for the 32 bits instructions. The test code used for the experiment is presented in Listing 3 with the corresponding execution duration for each word. The instructions contained in the same word have the same color.

VII. Results

A fault injection attack campaign has been conducted on the test code using the TRAITOR platform and the laser fault injection platform. The results are presented and compared below.

```
...                 ...
adds R0,R0,#9       2 cycles
subs R3,R3,#4
subs R2,R2,#13      2 cycles
adds R6, R6,#5
subw R5,R5,#1       1 cycle
addw R1,R1,#17      1 cycle
subw R0,R0,#19      1 cycle
adds R3,R3,#20      2 cycles
subs R2,R2,#24
subw R4,R4,#2       1 cycle
addw R0,R0,#31      1 cycle
adds R5,R5,#29      2 cycles
subs R6,R6,#41
subs R4,R4,#37      2 cycles
adds R3,R3,#51
adds R2,R2,#57      2 cycles
subs R1,R1,#61
addw R5,R5,#6       1 cycle
subs R6,R6,#44      2 cycles
adds R2,R2,#53
...                 ...
NOPS                2 cycles
```

Listing 3: Test code used for the GARLIC application with the execution duration for each word.

A. TRAITOR injection results

To perform the attack, we set the amplitude for TRAITOR glitches to 445 which makes possible the skip/repeat fault on every word of the test code regardless the instruction size. The results are plotted for the delays ranging from 61 to 74 cycles, respectively when the first and the last words are faulted. The attack was repeated 50 times with the same set of parameters giving the same results. In the following Figures, the color and index *n* of each word is related to the test code (Listing 3). The fault timing is defined as the delay when the word n is skipped.

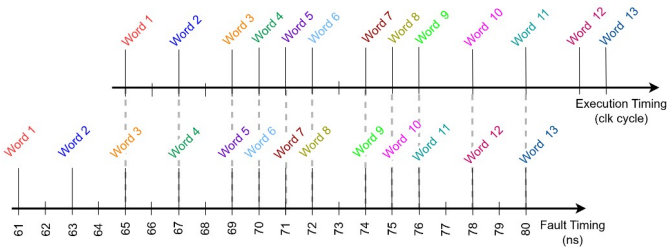


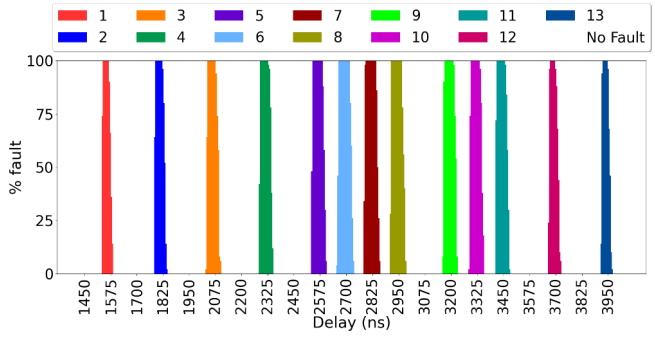
Figure 12: Execution timing and fault timing for each word when attacking the test code with TRAITOR.

In Figure 12 the execution timing is depicted at the top and the fault timing is represented at the bottom. First, we can observe the same pattern for both execution timing and fault timing. This means that for every fault we can identify the instruction word being executed. For example, after faulting the word 3 we need to wait 2 cycles to obtain a fault on the word 4. During that time a word is executed in 2 cycles. This

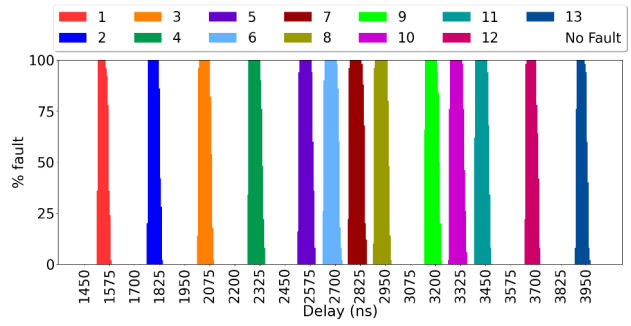
indicates that the fault occurs before the execution stage. By doing this comparison for every fault and as the execution timing pattern is non periodic, we can conclude that the fault on the word n occurs during the execution of the word (n-2). This supports the initial hypothesis that the skip/repeat is a disruption during the instruction transfer from the flash memory to the prefetch buffer.

B. Laser injection results

Two experiments have been performed using laser injection. We selected two bits to be faulted, one in the low part of the word (bit 18) and one in the high part of the word (bit 8) such that, with a word containing two instructions of 16 bits, faults on both instructions can be observed. The fault probability with respect to the delay is plotted in Figure 13a for the bit 8 and in Figure 13b for the bit 18. First, we can see that the fault timing is the same for both bits attacked. Second, we observe that the delay between each fault is approximately 125 ns or 250 ns which is equivalent to one or two times the clock period value. The alternation between 1 and 2 periods corresponds to the same pattern as the execution timing.



a. Faults on bit 8.



b. Faults on bit 18.

Figure 13: Percentage of faults obtain by attacking two different bits of each word at different delay.

The schematic of this pattern for the execution timing and for the laser fault timing is represented in Figure 14. We can see the presence of a delay between the execution pattern and the fault pattern. We can make the same observation that we previously made on the TRAITOR results: the fault appears on word n during the execution of the word (n-2).

C. Comparison and conclusions

When looking at the results of both fault injection campaigns with TRAITOR and with laser, we draw the same conclusion. In both cases we retrieve the same fault pattern that is shifted by 2 words. This means that the fault on the word n occurs during the execution of the word (n-2). Given this result, we can conclude that the fault occurs on the same stage while making fault injection with TRAITOR and with laser. As the laser injection is performed on the flash memory thus it disrupts the transfer from the flash memory to the prefetch buffer. Therefore, the analysis supports the initial assumption that the TRAITOR skip/repeat is due to the disruption of the transfer from the flash memory to the prefetch buffer.

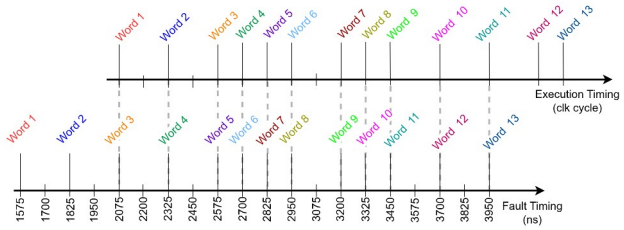


Figure 14: Execution timing and fault timing for laser attacking the test code.

D. Discussion

GARLIC is presented here as a proof of feasibility on a simple microcontroller for a well-defined fault model. The laser injection has been performed on the flash memory which is easy do identify on the microcontroller. To expand the method to other fault effects it is necessary to have the possibility of making fault injection on other parts (RAM memory, logic, registers). The RAM memory is also easily identified and considering the state of the art, it is now possible to obtain fault on registers using laser [20]. As RAM and registers are used in the execute stage, we could apply GARLIC on glitch effect that could happens during the execution.

Cortex-M microprocessors are commonly used to study fault injection because of their simple architecture with no countermeasures against physical attack [17] [19]. It makes possible the characterization of fault effects and the elaboration of complex attacks. Some works have been made on SoC showing that the presence of cache memory makes the study much more complicated [21], [22]. We could expect different effects of TRAITOR glitch in this case. As the cache memory is not fetch periodically the timing analysis could be complicated. Nevertheless we still could make two separate studies with and without cache activated and distinguish different fault effects [4].

VIII. Conclusion and Perspectives

The characterization of the effects observed when performing fault injection is made at ISA level. In particular, accessing to the mechanism of fault propagation is often challenging and leads to uncertain descriptions. Glitch fault injection is a non-located injection technique. In this case, the investigation over the fault propagation from the physical level to the software level is even more complicated. The GARLIC

method presented in this article aims to overcome this problem. This method is based on a comparison between glitch and laser fault timing. This method has been applied on `TRAITOR` clock glitch that induces skip/repeat. The `GARLIC` application on the skip/repeat effect obtained with `TRAITOR` supports the hypothesis that it is due to a disruption during the transfer of instructions from the flash memory to the prefetch buffer.

In further work, the method could be applied on other fault effects such as the ones observed during the `TRAITOR` characterization (section IV). `GARLIC` could also be applied to fault effects expected to occur during the decode or the execute stage considering the possibility of making fault injection on RAM memory and registers.

ACKNOWLEDGMENT

The authors thank the CYBER-ELEC platform with IETR (CNRS 6164) for laser fault injection facilities.

REFERENCES

[1] N. Beringuier-Boher, K. Gomina, D. Hély, J. Rigaud, V. Beroulle, A. Tria, J. Damiens, P. Gendrier, and P. Candelier, "Voltage glitch attacks on mixed-signal systems," in 17th Euromicro Conference on Digital System Design, DSD 2014, Verona, Italy, August 27-29, 2014. IEEE Computer Society, 2014, pp. 379–386.

[2] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, "Software fault resistance is futile: Effective single-glitch attacks," in 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016. IEEE Computer Society, 2016, pp. 47–58.

[3] L. Claudepierre, P.-Y. Péneau, D. Hardy, and E. Rohou, "TRAITOR: A low-cost evaluation platform for multifault injection," in ASSS '21: Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems, Virtual Event, Hong Kong, 7 June, 2021,

[4] L. Rivière, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer, and L. Sauvage, "High precision fault injections on the instruction cache of ARMv7-M architectures," in IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015. IEEE Computer Society, 2015, pp. 62–67.

[5] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures against fault attacks on software implemented AES: effectiveness and cost," in Proceedings of the 5th Workshop on Embedded Systems Security, WESS 2010, Scottsdale, AZ, USA, October 24, 2010. ACM, 2010, p. 7.

[6] M. Dumont, P. Moëllic, R. A. C. Viera, J. Dutertre, and R. Bernhard, "An overview of laser injection against embedded neural network models," in 7th IEEE World Forum on Internet of Things, WF-IoT 2021, New Orleans, LA, USA, June 14 - July 31, 2021. IEEE, 2021, pp. 616–621.

[7] M. Dumont, M. Lisart, and P. Maurine, "Modeling and simulating electromagnetic fault injection," IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., vol. 40, no. 4, pp. 680–693, 2021.

[8] I. Alshaer, G. Burghoorn, B. Colombier, C. Deleuze, V. Beroulle, and P. Maistri, "Cross-layer analysis of clock glitch fault injection while fetching variable-length instructions," J. Cryptogr. Eng., vol. 14, no. 2, pp. 325–342, 2024.

[9] J. Laurent, C. Deleuze, F. Pebay-Peyroula, and V. Beroulle, "Bridging the gap between RTL and software fault injection," ACM J. Emerg. Technol. Comput. Syst., vol. 17, no. 3, pp. 38:1–38:24, 2021.

[10] M. S. Kelly, K. Mayes, and J. F. Walker, "Characterising a CPU fault attack model via run-time data analysis," in 2017 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2017, McLean, VA, USA, May 1-5, 2017. IEEE Computer Society, 2017, pp. 79–84.

[11] Jonah Alle Monne, Guillaume Bouffard, "Synthesis of rtl-based characterization programs for fault injection," in IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2026, Washington, USA, May 4-7, 2026, 2026.

[12] T. Trouchkine, G. Bouffard, and J. Clédière, "Fault injection characterization on modern cpus," in Information Security Theory and Practice - 13th IFIP WG 11.2 International Conference, WISTP 2019, Paris, France, December 11-12, 2019, Proceedings, ser. Lecture Notes in Computer Science, M. Laurent and T. Giannetsos, Eds., vol. 12024. Springer, 2019, pp. 123–138.

[13] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller," in 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013, W. Fischer and J.-M. Schmidt, Eds. IEEE Computer Society, 2013, pp. 77–88.

[14] A. Gicquel, L. Claudepierre, D. Hardy, K. Heydemann, and E. Rohou, "Chapati: End-to-end methodology to conduct multi-faults injection campaigns," in IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2026, Washington, USA, May 4-7, 2026.

[15] C. O'Flynn and Z. D. Chen, "Chipwhisperer: An open-source platform for hardware embedded security research," in Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers, ser. Lecture Notes in Computer Science, E. Prouff, Ed., vol. 8622. Springer, 2014, pp. 243–260.

[16] J. Balasch, B. Gierlichs, and I. Verbauwhede, "An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus," in 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, DTC 2011, Tokyo, Japan, September 29, 2011, L. Breveglieri, S. Guilley, I. Koren, D. Naccache, and J. Takahashi, Eds. IEEE Computer Society, 2011, pp. 105–114.

[17] I. Alshaer, B. Colombier, C. Deleuze, V. Beroulle, and P. Maistri, "Variable-length instruction set: Feature or bug?" in 25th Euromicro Conference on Digital System Design, DSD 2022, Maspalomas, Spain, August 31 - Sept. 2, 2022. IEEE, 2022, pp. 464–471.

[18] A. Marotta, R. Lashermes, G. Bouffard, O. Sentieys, and R. Dafali, "Characterizing and modeling synchronous clock-glitch fault injection," in Constructive Side-Channel Analysis and Secure Design - 15th International Workshop, COSADE 2024, Gardanne, France, April 9-10, 2024, Proceedings, ser. Lecture Notes in Computer Science, R. Wacquez and N. Homma, Eds., vol. 14595. Springer, 2024, pp. 3–21.

[19] B. Colombier, A. Menu, J.-M. Dutertre, P.-A. Moëllic, J.-B. Rigaud, and J.-L. Danger, "Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller," in IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2019, McLean, VA, USA, May 5-10, 2019. IEEE, 2019, pp.1–10.

[20] H. Perrin, J. Dutertre, and J. Rigaud, "Betrayed by light: How photon emission microscopy empowers register bit-level laser attacks on microcontrollers," in IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2025, San Jose, CA, USA, May 5-8, 2025, pp. 35–45.

[21] T. Trouchkine, S. K. Bukasa, M. Escouteloup, R. Lashermes, and G. Bouffard, "Electromagnetic fault injection against a system-on-chip, toward new micro-architectural fault models," CoRR, vol. Abs/1910.11566, 2021.

[22] S. Casavecchia, D. Aboulkassimi, J. Clédière, J.-M. Dutertre, and S. Pontié, "Exploring laser fault injection in system-on-chips: A new approach for cpu and cache attacks," in IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2026, Washington, USA, May 4-7, 2026.

Dolunay: Architectural Support for Independent Thread Scheduling in a RISC-V SIMT Accelerator

Ahmet Zahit Can
Computer Engineering, A. I. and Data Engineering
Yıldız Technical University
Istanbul, Türkiye
ahmetzahit.can@yildiz.edu.tr

Erkan Uslu
Computer Engineering
Yıldız Technical University
Istanbul, Türkiye
euslu@yildiz.edu.tr

Abstract— **Single-Instruction Multiple-Thread (SIMT) architectures have revolutionized data-parallel computing by providing a high-throughput abstraction that simplifies vector management. However, traditional stack-based SIMT models do not support intra-warp synchronization primitives such as mutexes and spin-locks. This work introduces Dolunay, a RISC-V-based Independent Thread Scheduling (ITS) SIMT accelerator. By only adding three custom instructions, Dolunay employs a cooperative multitasking model and explicit synchronization barriers at the hardware-level, and provides the forward-progress guarantees necessary to implement starvation-free algorithms. We evaluate Dolunay on the Cmod A7-35T FPGA module and demonstrate its ability to correctly execute kernels that deadlock on traditional stack-based architectures while still achieving parallel execution for conventional compute-heavy kernels.**

Keywords-Independent Thread Scheduling, SIMT, GPGPU, RISC-V, Forward-progress guarantee, Barrier synchronization, Cooperative multitasking

I. INTRODUCTION

Popularized by NVIDIA [1], Single-Instruction Multiple-Thread (SIMT) architectures employ lockstep execution, a model in which a single instruction stream drives multiple independent threads simultaneously [1], [2]. These threads are logically grouped into a warp; at the microarchitectural level, this organization allows individual threads to occupy SIMD lanes while the control logic operates at warp granularity. This abstraction maximizes computational throughput without imposing the burden of manual vector management on the programmer [1], [3]. Consequently, the SIMT model has become the architectural foundation of General-Purpose GPU (GPGPU) computing and has proven highly effective across a broad range of data-parallel workloads [4].

However, NVIDIA's pre-Volta SIMT designs were unable to support intra-warp synchronization primitives such as mutexes. To address this limitation, NVIDIA introduced Independent Thread Scheduling (ITS) in the Volta architecture, thereby enabling support for such primitives [5], [6].

The SIMT design philosophy has also been applied to RISC-V in several implementations, bringing SIMT computing to the RISC-V ecosystem [7]–[10]. However, none of these designs provide sufficient forward-progress guarantees required to support the synchronization primitives discussed above. This work introduces Dolunay, a RISC-V-based ITS SIMT accelerator. Each path in the Dolunay architecture has its own PC value, and the RISC-V ISA is extended with a custom extension that supports cooperative multitasking and explicit reconvergence via synchronization barriers, thereby enabling a variety of synchronization primitives. Consequently, starvation-free algorithms can be implemented on Dolunay. To the best of our knowledge, this is the first RISC-V SIMT accelerator to provide this capability. The source code for Dolunay is published at https://github.com/ahmetzahitcan/dolunay, licensed under Apache License 2.0. The exact variant described in this paper is found under branch `arch__dsd2026`.

II. RELATED WORK

SIMT architectures organize threads into warps, with each warp sharing a single instruction stream. A primary challenge in such architectures is branch divergence, in which conditional branches cause threads within a warp to follow different execution paths. Because the hardware employs a shared control unit, it must serialize these paths by executing them sequentially while masking lanes that are inactive on the current path. Ideally, once the paths reconverge at a common instruction, all lanes are unmasked to restore full SIMD throughput [1], [2], [4], [11].

In pre-Volta NVIDIA architectures [5], reconvergence was managed using an Immediate Post-Dominator (IPDOM) stack model. This mechanism uses a hardware stack to track divergence events; because nested branches may occur before an earlier branch reconverges, the stack preserves the execution state associated with each level of divergence. Each stack entry typically stores the active mask for a divergent path together with the address of its reconvergence point, that is, the IPDOM. When execution reaches a reconvergence point, the corresponding entry is popped and the previous mask is restored, allowing the warp to resume unified execution [1], [11]. Beyond NVIDIA architectures, this model has also been implemented in several RISC-V SIMT accelerators, including Vortex [7] and e-GPU [8].

However, a significant limitation of the stack-based model is its inability to support intra-warp synchronization primitives,

Manuscript received May 11, 2026; revised August 21, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.137

such as mutexes and spin-locks. Because divergent paths execute serially under a shared warp execution context, a thread holding a lock may be inactive while another thread in the same warp spins while waiting for it, resulting in deadlock [5]. To address this limitation, NVIDIA transitioned to ITS beginning with the Volta architecture [5]. In the ITS model, threads maintain independent execution state, including separate program counters and call stacks, which allows the scheduler to make forward-progress decisions at thread granularity [5], [6].

This design allows the hardware to schedule threads more flexibly by dynamically grouping those with matching PCs while avoiding the lockstep deadlock scenarios associated with stack-based SIMT execution. The shift fundamentally strengthens the programming model by enabling fine-grained synchronization and more complex data structures within a single warp [5].

Beyond stack-based and ITS models, a third approach is transparent, or microarchitectural, SIMT, in which SIMT execution is implemented entirely below the ISA level. This approach is exemplified by Simty [9], SIMTIGHT [10], and SIMT-X [12]. It relies on dynamic hardware detection of divergence and reconvergence without requiring explicit software annotations, thereby enabling the SIMT execution of general-purpose binaries [9]. Although this approach shares the use of per-thread PCs with the ITS model, the aforementioned implementations do not support the synchronization primitives discussed above.

Dolunay was initially inspired by transparent SIMT architectures. Early designs explored combining implicit hardware reconvergence with preemptive scheduling to support intra-warp forward progress. However, efficiently detecting reconvergence among independently scheduled execution contexts required substantial additional hardware complexity, and unconstrained scheduling policies could lead to suboptimal reconvergence rates, thereby incurring significant performance overhead. Consequently, Dolunay adopts an architectural ITS approach based on cooperative scheduling and explicit synchronization barriers for reconvergence.

III. Methodology

A. Instruction Set

Dolunay implements a variety of standard extensions as well as the custom Xdolunay extension on top of the RV32I base.

TABLE I. Xdolunay custom instructions

Instruction	Encoding	Description
`BINIT imm(rs1)`	I-format (opcode = `0xB`, funct3 = `1`, rd = `x0`)	Initializes a barrier in an aligned 4-byte region of work RAM.
`BSYNC imm(rs1)`	I-format (opcode = `0xB`, funct3 = 2, rd = `x0`)	Updates the barrier in an aligned 4-byte region of work RAM and either parks the calling threads or releases the barrier.
`YIELD`	Word (`0x0000400B`)	Yields control to another path.

The standard extensions Zicsr and Zalrsc are mandatory. Xdolunay depends on Zicsr for custom CSRs, while Zalrsc provides the LR/SC operations required to implement synchronization primitives. Although Xdolunay supports independent thread execution with per-thread program counters, it does not introduce synchronization primitives beyond those already provided by Zalrsc.

The standard extensions Zicond and Zba were included to improve performance for operations expected to be common in Dolunay's target workloads. Zicond reduces the need for short conditional branches, mitigating some forms of divergence and thereby reducing, though not eliminating, the need for explicit convergence through barriers, as detailed in Section III-A1. Zba improves address-generation efficiency for accesses indexed by warp and/or thread identifiers.

The standard extensions Zicntr and Zihpm are used for performance monitoring. The hardware performance monitors (HPMs) are described in detail in Section III-A3.

1) Custom Instructions

RISC-V SIMT architectures differ in whether their divergence and reconvergence mechanisms are implicit or explicit. In some architectures, such as Simty [9] and SIMTIGHT [10], these mechanisms are implicit, meaning that they require no software annotation. Divergence occurs automatically on non-uniform branch and jump instructions. For reconvergence, paths are sorted by priority, and reconvergence occurs when the active path (with the highest priority) and the next path (with the second-highest priority) are detected to have the same PC. The priority-sorting mechanism ensures that this condition is sufficient to cover all reconvergence cases. Together, these features enable microarchitectural SIMT, allowing the hardware to execute traditional SPMD programs in SIMT fashion [9].

In other architectures, such as Vortex [7] and e-GPU [8], these mechanisms are explicit and must be marked using custom instructions, either by the programmer or the compiler. Failure to mark these points may result in incorrect execution. Vortex and e-GPU both use the same custom extension [8], which supports an IPDOM-stack-based approach in which all threads within a warp share a single PC. These custom instructions manipulate the IPDOM stack and the active thread mask, and all branches and jumps are assumed to be uniform across the active thread mask [7].

Dolunay adopts an intermediate approach: divergence is implicit, whereas reconvergence is explicit. Similar to Simty [9] and SIMTight [10], Dolunay makes no assumption of uniformity for branches and jumps and automatically splits paths when a non-uniform branch or jump is detected. However, because Dolunay allows threads to yield control to one another, it has no notion of path priority. As a result, implicit reconvergence detection would require checking whether the active path shares a PC with any inactive path, which would introduce significant hardware complexity. Therefore, Dolunay requires all reconvergence points to be explicitly annotated in software using custom instructions.

To achieve this, Dolunay adds 3 custom instructions: BINIT, BSYNC and YIELD. Encoding and a brief description for these instructions is given in Table I.

Dolunay uses warp-level structures called barrier contexts to support reconvergence via synchronization barriers. Barrier contexts are stored in memory, and both BINIT and BSYNC therefore perform memory accesses. Unlike branches and jumps, Dolunay assumes uniformity for barrier context addresses; violating this assumption results in undefined behavior. Storing the barrier contexts in memory was preferred over dedicated registers to avoid the need to spill and restore warp-level state. As described in Section III-C, the programming model uses unmodified RISC-V GCC and inserts the custom instructions via .insn assembler directives. Because the toolchain is not warp-aware, spilling and restoring such registers would be particularly cumbersome.

The barrier context contains two thread masks. The first is the participation mask. For each bit, a value of 1 indicates that the corresponding thread participates in the synchronization barrier and that the other participating threads must wait for it, whereas a value of 0 indicates that the thread does not participate in that barrier. The second is the parked mask. For each bit, a value of 1 indicates that the corresponding thread has parked at the barrier and is waiting for the remaining participating threads, whereas a value of 0 indicates that the thread has either not yet parked at that barrier, if its participation bit is 1, or does not participate in that barrier, if its participation bit is 0.

BINIT initializes the barrier context. The participation mask is set to the active thread mask, meaning that synchronization applies to the threads that execute BINIT simultaneously. The parked mask is initialized to all zeros.

BSYNC performs synchronization using the barrier context. Its operation proceeds as follows:

1) **Compute the new barrier context:** The new parked mask is computed as the bitwise OR of the active thread mask and the previous parked mask. This value is always written back to memory.
2) **Evaluate the new barrier context:** The new parked mask is compared with the participation mask. Depend- ing on the result, either step 3 or step 4 is executed.
3) **Release the barrier:** If the condition in step 2 evaluates to true, all parked threads are merged into the current path.
4) **Park the threads:** If the condition in step 2 evaluates to false, the current path is deactivated, and the next active path is selected using the round-robin scheduler, which is the same mechanism used by YIELD.

The algorithm for BSYNC is given in Fig. 1. A notable consequence of this implementation is that the reconvergence point itself is not stored in the barrier context. Instead, it is encoded implicitly by the location of BSYNC in the instruction stream. One implication is that if multiple BSYNC instructions use the same barrier context, an individual thread may park at one BSYNC instruction and later resume at another. This behavior is not currently used by any of the software developed for this architecture.

1: **procedure** BSYNC(addr)
2: ▷ T is the number of threads per warp
3: $m_{parked} \leftarrow MEM[addr][T-1:0]$
4: $m_{participation} \leftarrow MEM[addr][2T-1:T]$
5: $m_{parked\_new} \leftarrow m_{parked} \vee m_{active}$
6: **if** $m_{parked\_new} = m_{participation}$ **then**
7: $m_{active} \leftarrow m_{participation}$
8: **else**
9: $m_{active} \leftarrow 0$
10: YIELD
11: **end if**
12: $MEM[addr][T-1:0] \leftarrow m_{parked\_new}$
13: **end procedure**

Figure 1. BSYNC algorithm

YIELD switches the active path using a round-robin scheduler.

2) CSRs

In addition to the three custom instructions, Xdolunay defines three custom CSRs. Their names, addresses, and brief descriptions are given in Table II.

All three CSRs are read-only. `xwarpid` and `xthrid` are constant for a given thread. The standard CSR `mhartid` is also implemented. The lower bits of `mhartid` are equal to `xthrid`, and the remaining bits are equal to `xwarpid`.

`xrole` can be used to determine whether the current threadis the leader of the active path. The leader is the thread with the lowest ID among those enabled in the active thread mask. Every path has exactly one leader and zero or more follower threads.

All other CSRs are implemented as read-only zero.

3) Hardware Performance Monitors

Dolunay implements the three standard HPMs together with two custom HPMs. The standard HPMs `cycle` and `time` are implemented as shadows and are read from a counter that increments every cycle. The standard HPM `instret` indicates the number of instructions retired by the current thread.

TABLE II. XDOLUNAY CUSTOM CSRS

CSR	Address	Description
`xwarpid`	`0xCC0`	Warp identifier
`xthrid`	`0xCC1`	Thread identifier within the warp. Threads from different warps may have the same xthrid value.
`xrole`	`0xCC2`	Indicates whether the thread is a leader (0) or a follower (1).

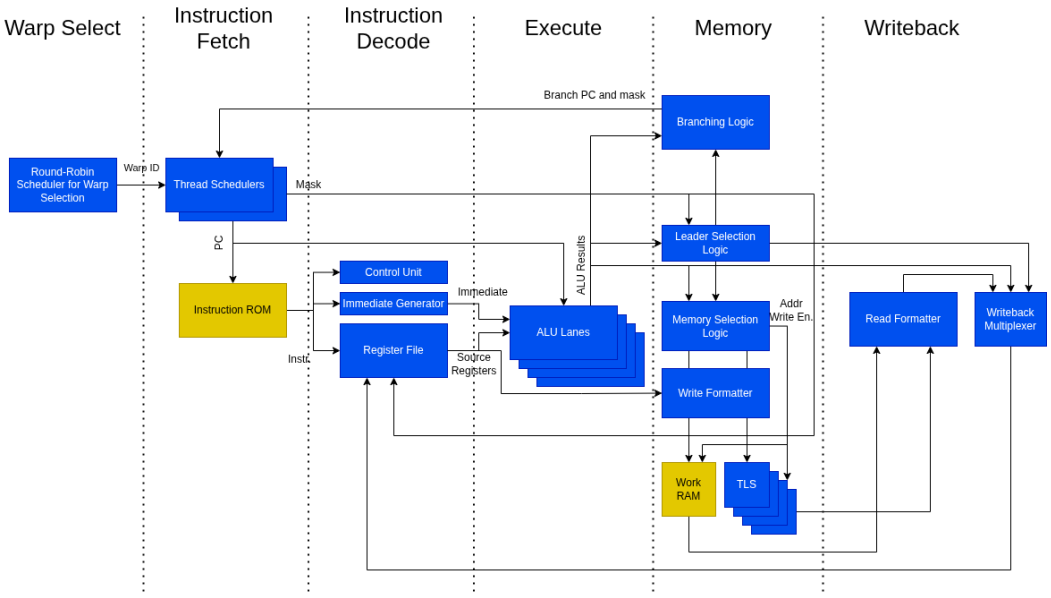


Figure 2. Dolunay core. Yellow modules are external to the core.

The custom HPM `wtinstret`, mapped to `hpmcounter4`, is a warp-level counter that indicates the total number of instructions executed by the warp. It is always equal to the sum of the `instret` values of all threads within the warp.

The custom HPM `wuinstret`, mapped to `hpmcounter3`, is a warp-level counter that indicates the number of instructions executed uniquely by the warp. Simultaneous executions of an instruction are counted as a single unique execution of that instruction. In the absence of divergence, `wuinstret` is equal to `wtinstret` divided by the number of threads warp.

B. Microarchitecture

Similar to other SIMT architectures [1], Dolunay is designed to execute multiple threads in SIMD fashion. A group of co-executing threads is referred to as a warp. Dolunay also interleaves the execution of multiple warps. As detailed in Section III-B4, four is the minimum supported number of warps, although a larger number may also be selected before synthesis. Likewise, the number of threads per warp can be chosen before synthesis, provided that all warps use the same thread count.

As the threads in a warp diverge, they are divided into subgroups called paths. As per the ITS design, paths run using a cooperative multitasking design, allowing the successful execution of starvation-free algorithms. Paths are managed by the thread schedulers, the number of thread schedulers is equal to the number of warps. The implementation of the thread schedulers is detailed in Section III-B2.

A simplified overview of the core is given in Fig. 2.

1) Pipeline

The pipeline has 6 stages: Warp Select, Instruction Fetch, Instruction Decode, Execute, Memory and Writeback.

1. **Warp Select:** This stage contains a round-robin scheduler that selects the next warp to run.
2. **Instruction Fetch:** During this stage, the relevant thread scheduler is run, selecting a path to execute. The PC of the selected path is sent to the Instruction ROM to fetch the instruction. The mask of the selected path is also pushed into the pipeline.
3. **Instruction Decode:** This stage contains the control unit. The control unit generates the control signals based on the instruction read from the ROM. The source register indices are simultaneously used to read from the register file. The register file is synchronous and provides two read ports and one write port. To implement this structure using FPGA BRAMs, we employ a shadowing technique in which two BRAMs, each with one read port and one write port, share a common write interface.
4. **Execute:** This stage contains the ALUs for computation. The number of ALU lanes is equal to the number of threads per warp.
5. **Memory:** This stage contains several units running in parallel. As the name implies, memory accesses happen in this stage. The leader selection and coalescing logic used for memory accesses are reused for JALR and branching. Divergence is detected and the relevant thread scheduler is notified, updating its paths. Instruction retired and instruction replay signals are also generated in this stage, consumed by the relevant thread scheduler and the hardware performance monitors.

6. **Writeback:** This stage is used for writing the results into the register file.

2) Thread Schedulers

Thread schedulers are hardware units that manage thread divergence and convergence. Each warp is assigned a dedicated scheduler, which groups threads into paths. A path is a simple data structure that contains a PC and a thread mask with at least one active thread. At startup, there is only one path, which contains all threads and has a PC value of zero. Divergence increases the number of paths, up to a maximum equal to the number of threads per warp, corresponding to the worst-case scenario in which each thread has a different PC. Reconvergence decreases the number of paths, thereby allowing more threads to execute in parallel.

It is worth noting that, at the microarchitectural level, Dolunay does not assign a unique PC to each thread, but rather to each path. However, because Dolunay correctly handles the worst-case scenario of one thread per path, this distinction is not architecturally visible.

Upon receiving a branch signal, the scheduler determines whether divergence has occurred. If the branch is non uniform, the threads that take the branch are partitioned into a new path. For uniform branches, in which all active threads follow the same trajectory, the PC of the current path is simply updated.

As explained in Section III-A1, Dolunay allows software to switch to a different path. This is achieved by a yield signal generated when the YIELD custom instruction reaches the Memory stage. The signal is qualified by the warp ID associated with that stage. When the signal is asserted, the scheduler performs a context switch to the next active path within the warp, thereby enabling algorithms that require a forward-progress guarantee, which is a key focus of this work. The next active path is selected by a modified round-robin scheduler that skips inactive paths.

The scheduler receives two input signals related to instruction completion: an instruction-complete signal and an instruction replay mask. When the instruction-complete signal is asserted, the replay mask is checked to determine whether any threads must replay the instruction. If so, the instruction is replayed using a mask that includes only those threads; otherwise, the PC of the active path is incremented.

The BSYNC instruction, described in Section III-A1, executes in two passes. A single-bit internal register is maintained for each warp to track the current pass. In the first pass, the barrier context is loaded from memory into an internal register. In the second pass, that register is used either to park the threads or to release the barrier. In both cases, the active thread mask is updated, either to zero or to the participation mask stored in the barrier context. In the parking case, control is additionally yielded to another path.

3) Memory

Dolunay separates the address space into three regions: Instruction ROM (IROM), Work RAM (WRAM), and Thread-Local Storage (TLS). The corresponding address ranges for these three regions are given in Table III.

TABLE III. MEMORY REGIONS

Memory Region	Start Address	End Address
IROM	`0x00000000`	`0x3FFFFFFF`
WRAM	`0x40000000`	`0x7FFFFFFF`
TLS	`0x80000000`	`0xFFFFFFFF`

To determine which memory region is being accessed, the pipeline checks the highest two bits of the address. Although the current configuration allocates 1 GiB to IROM, 1 GiB to WRAM, and 2 GiB to TLS, the corresponding memory units are not large enough to cover the full ranges. Therefore, the memory is effectively shadowed because the upper address bits are ignored.

WRAM has a 32-bit-wide read-write port with byte-write enable. For each access to WRAM, a leader thread is selected to provide the address. The leader's operation always succeeds. Any follower thread whose address differs from the leader's replays the operation. Any follower thread whose address matches the leader's completes the operation simultaneously with the leader. For load operations, this means that such threads read the same value as the leader. For store operations, writes from non-replayed follower threads are ignored. This choice is architecturally valid because it is indistinguishable from the case in which the leader is the last thread to perform its store.

IROM has two 32-bit-wide read-only ports. One of these ports is used to fetch instructions, and only the IROM region is executable. The other port is used to load constants. As with WRAM, a leader thread is selected to provide the address. The leader's load operation always succeeds. Any follower thread with a matching address completes its load simultaneously with the leader. Other follower threads replay. Stores to IROM are ignored.

TLS instances are per-thread read-write memories. Each execution lane has an independent TLS with a 32-bit-wide read-write port with byte-write enable. The warp ID is appended to the address to ensure that threads with the same thread ID in different warps do not access one another's TLS. TLS accesses always complete in a single pass.

Because different threads can generate different addresses, different threads may access different regions. IROM and WRAM accesses use the same leader-selection logic, which selects only one leader, so only one of IROM or WRAM can be accessed in a single cycle. Threads accessing TLS are exempt from leader selection.

LR/SC operations are valid only on WRAM. LR reserves the entire WRAM region, and SC invalidates the reservations held by other threads. When multiple threads execute SC simultaneously, a leader thread is selected to perform the operation. All other threads report failure by storing 1 in `rd`. SC instructions never replay.

This memory design, although simple, is inefficient. As shown in Section IV-B, the extensive serialization of memory

accesses significantly degrades performance. We explored more efficient memory implementations, but implementing them within the limited area and BRAM resources of the Cmod A7-35T proved cumbersome. We therefore chose this simpler, less efficient implementation, as the primary goal of Dolunay is to serve as a proof of concept for a RISC-V ITS SIMT architecture rather than as a high-performance soft-core accelerator.

4) FGMT

Dolunay avoids the need for complex dependency-tracking hardware by employing Fine-Grain Multithreading (FGMT), which interleaves instructions from different warps [13]. Although the Dolunay pipeline has six stages, four warps are sufficient to ensure correct execution. The argument is as follows.

Consider instruction P entering the pipeline at cycle N . Because the pipeline issues another instruction from the same warp only after W cycles, where W is the number of warps, instruction P + 1 enters the pipeline at cycle N + W . The progression of these two instructions through the pipeline, together with the corresponding cycle numbers, is shown in Table IV.

The following hazards must be avoided:

1. **Hazard #1 — Register-file write before read:** The register write of instruction P at the Writeback stage, which occurs in cycle N + 5, must complete before the register read of instruction P + 1 at the Instruction Decode stage, which occurs in cycle N + W + 2.
2. **Hazard #2 — Thread-scheduler state update:** The thread-scheduler state update of instruction P at the Memory stage, which occurs in cycle N + 4, must complete before the thread scheduler generates the PC and mask for instruction P + 1 at the Instruction Fetch stage, which occurs in cycle N + W + 1.
3. **Hazard #3 — BSYNC toggle:** As explained in Section III-B2, the BSYNC instruction executes in two passes, and a register is toggled to track the current pass. This register must be updated at the Memory stage in cycle N + 4 before the corresponding control signals are generated at the Instruction Decode stage in cycle N + W + 2.

Therefore, the following inequalities must hold:

$$N + W + 2 > N + 5 \Rightarrow W > 3$$

$$N + W + 1 > N + 4 \Rightarrow W > 3$$

$$N + W + 2 > N + 4 \Rightarrow W > 2$$

TABLE IV. PIPELINE CYCLE NUMBERS FOR INSTRUCTIONS

Inst.	WS	IF	ID	EX	MEM	WB
P	N	N+1	N+2	N+3	N+4	N+5
P+1	N+W	N+W+1	N+W+2	N+W+3	N+W+4	N+W+5

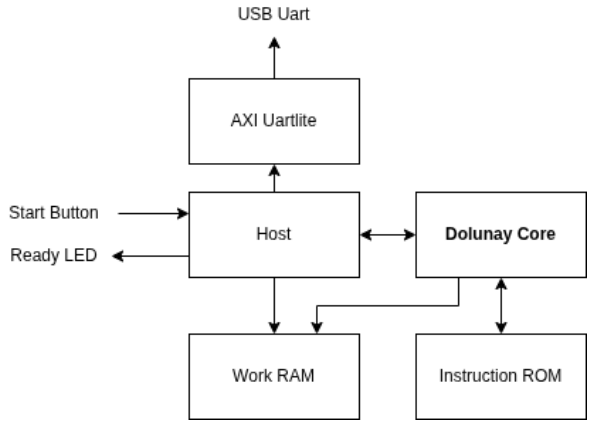


Figure 3. Testbench organization and hardware integration on the Cmod A7-35T FPGA.

Hence, W ≥ 4 is required to avoid hazards. When W = 3, hazards #1 and #2 occur; when W < 3, all three hazards occur.

C. Programming Model

For the programming model, we implemented a minimal C toolchain. Custom instructions are inserted into C code via asm directives, and because the assembler is also unaware of these custom instructions, .insn directives are used to instantiate them. A simple linker script maps ELF sections to memory regions, and a crt0.s file initializes the stack pointer, copies globals from IROM to WRAM or TLS, and initializes two BSS sections, one in WRAM and the other in TLS. We used unmodified RISC-V GCC to generate ELF objects and converted them to binaries using the objdump tool.

IV. RESULTS

To test the core, we connect it to a simple FSM host via a start/ready mechanism. The FSM host also drives AXI Uartlite llowing to print messages and dump memory. A diagram of the design is given in Fig. 3. This design as a whole is then synthesized and implemented for Cmod A7-35T using Vivado 025.2 toolchain. Flow PerfOptimized high and Performance Explore strategies are used.

A. Area and Frequency

We synthesized our testbench module with warp counts of four, eight, and sixteen; thread-per-warp counts of four, six, eight, and twelve; and TLS sizes of 512, 1024, 2048 and 4096 bytes. To name the various configurations, we use the following three-character format:

1. **v vs V vs W:** v indicates 4 warps, V indicates 8 warps, W indicates 16 warps.
2. **t vs s vs T vs S:** t indicates 4 threads per warp, s indicates 6 threads per warp, T indicates 8 threads per warp, S indicates 12 threads per warp. Admittedly, thread-per-warp counts of six and twelve are problematic because these values are not powers of two. However, higher powers of two

failed to implement, so we evaluated these configurations to examine how area scales with the number of threads per warp.

3. **h vs 1 vs 2 vs 4:** h indicates 512 bytes per TLS, 1 indicates 1024 bytes per TLS, 2 indicates 2048 bytes per TLS, 4 indicates 4096 bytes per TLS.

The post-implementation reported LUT, FF and BRAM usages as well as WNS and Fmax for each successful combination is given in Table V. All configurations were implemented with a 50MHz clock rate, Fmax is theoretical and calculated from WNS. Graphs for area scaling according to warp/thread configuration, for TLS size of 512 bytes, are given in Fig. 4.

We also tested whether changing the base ISA to RV32E could reduce BRAM usage by shrinking the register file; however, this was not beneficial except at extremely high warp counts. Because increasing the warp count raises area usage without improving IPC, we chose RV32I instead.

We selected the vT2 configuration for the performance benchmarks. We increased the IROM size to 8 KiB and the WRAM size to 32 KiB, which fully utilized the BRAM resources of the Cmod A7-35T in this configuration. Compute-heavy Kernels

We used four compute-heavy kernels to evaluate Dolunay: vecadd, saxpy, nearn, and gemm. For each warp, we extracted the wtinstret and wuinstret values described in Section III-A3 and calculated their ratio. This ratio is referred to as Lane Utilization Ratio, or LUR, and the obtained values are shown in Fig. 5a.

TABLE V. POST-IMPLEMENTATION RESOURCE USAGE AND PERFORMANCE CHARASTERICTICS ACROSS DOLUNAY CONFIGURATIONS

Config.	LUT	FF	BRAM	WNS	Fmax
vth	5594	3161	8	2.983ns	58.76MHz
vt1	5597	3161	10	2.446ns	56.96MHz
vt2	5599	3161	14	3.808ns	61.75MHz
vt4	5600	3161	22	3.333ns	59.99MHz
vsh	8746	4237	11	2.194ns	56.16MHz
vs1	8747	4237	14	1.709ns	54.67MHz
vs2	8742	4237	20	3.037ns	58.95MHz
vs4	8751	4235	32	2.612ns	57.51MHz
vTh	11167	5335	14	2.444ns	56.96MHz
vT1	11180	5335	18	2.922ns	58.55MHz
vT2	11230	5337	26	2.955ns	58.66MHz
vT4	11221	5339	42	1.758ns	54.81MHz
vSh	16924	7633	20	0.898ns	52.35MHz
vS1	19634	7633	26	1.124ns	52.97MHz
vS2	17007	7637	38	0.514ns	51.31MHz
Vth	7273	5336	10	2.126ns	55.94MHz
Vt1	7274	5336	14	1.883ns	55.19MHz
Vt2	7286	5336	22	3.108ns	59.19MHz
Vt4	7283	5345	38	2.722ns	57.87MHz
Vsh	11267	7287	14	1.989ns	55.52MHz
Vs1	11369	7308	20	0.948ns	52.48MHz
Vs2	11367	7281	32	1.763ns	54.83MHz
VTh	14412	9282	18	2.517ns	57.19MHz
VT1	14452	9284	26	1.869ns	55.15MHz
VT2	14436	9288	42	2.504ns	57.15MHz
Wth	10440	9667	14	2.291ns	56.46MHz
Wt1	10433	9671	22	2.002ns	55.56MHz
Wt2	10440	9675	38	1.424ns	53.82MHz
Wsh	16220	13410	20	0.638ns	51.64MHz
Ws1	16215	13416	32	1.400ns	53.76MHz

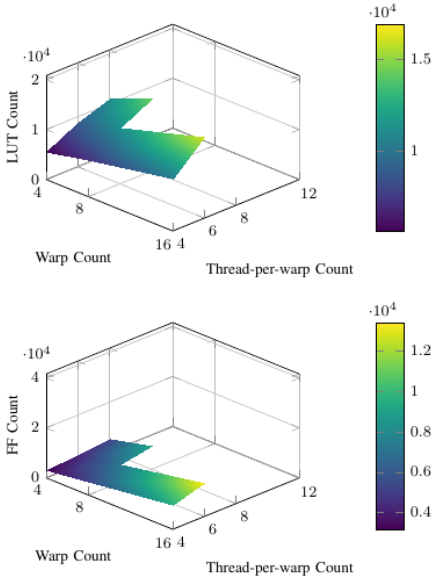


Figure 4. Scaling of LUT and FF counts according to warp/thread configuration

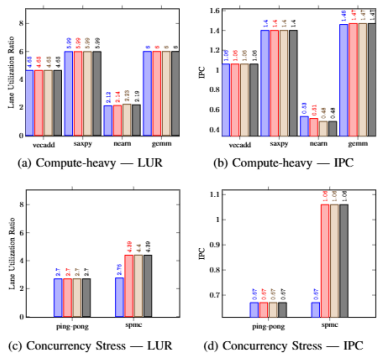
We also calculated the instructions-per-cycle value by dividing `wtinstret` by the cycle count. The cycle count was obtained by measuring the time required for Dolunay to assert ready after receiving the start signal, and it may therefore differ from the cycle counter. The resulting values are shown in Fig. 5b.

As explained in Section III-B3, we were unable to implement an efficient memory system because of the limitations of the Cmod A7-35T. All non-uniform, non-TLS memory accesses must be serialized, which significantly reduces IPC.

B. Concurrecy Stress Kernels

To demonstrate that Dolunay can correctly execute kernels that require a forward-progress guarantee, we designed two microkernels that cannot run on IPDOM-stack-based SIMT hardware.

The first is called ping-pong. Threads are grouped into pairs that share an accumulator; one thread executes the ping function, and the other executes the pong function. Both functions contain a loop that terminates once the accumulator reaches a predetermined value, which is 128 in our test case. Within the loop, both functions check whether the accumulator is currently even or odd. The ping function increments it by one if it is even and yields if it is odd. The pong function increments it by one if it is odd and yields if it is even. Without yielding, both functions would loop indefinitely while waiting for the value to change. Therefore, Dolunay's ability to complete this microkernel demonstrates that it can yield control between diverging threads.



(a) Compute-heavy — LUR (b) Compute-heavy — IPC

(c) Concurrency Stress — LUR (d) Concurrency Stress — IPC

Figure 5. Performance metrics for all kernels

The second is called spmc, which stands for Single-Producer Many-Consumers. One thread, selected globally across all warps rather than per warp, acts as the producer and reads values from an array into a shared variable. The consumer threads copy the value in the shared variable into their own arrays, thereby creating copies of the original array. The producer thread must wait until all consumer threads have completed their reads before it can write the next value, and all consumer threads must wait for the producer to write the data. This microkernel is intended to stress-test Dolunay's synchronization capabilities.

As above, we measured the LUR and IPC values, which are shown in Fig. 5c and Fig. 5d. Although both kernels ran to completion, they exhibited lower performance than the compute-heavy kernels. This is expected, because compute-heavy kernels rely on parallelism to utilize as many ALU lanes as possible. Indeed, the ping-pong kernel is limited to a maximum lane-utilization rate of four even in the best-case scenario, because in each warp half of the threads execute the ping function and the other half execute the pong function. The other kernel, spmc, showed that the warp containing the producer thread, which therefore executes two roles at the same time, performed significantly worse than the all-consumer warps, thereby clearly demonstrating the effect of running heterogeneous kernels on SIMT performance.

V. Conclusion

In this paper, we presented Dolunay, a RISC-V SIMT accelerator designed to overcome the synchronization limitations of traditional stack-based models. By implementing an Independent Thread Scheduling (ITS) architecture supported by three custom instructions, we demonstrated that cooperative scheduling and explicit barrier-based synchronization can provide an intra-warp forward-progress guarantee. Our results show that Dolunay successfully executes kernels that rely on this guarantee.

The Dolunay core presented here is primarily intended as a proof of concept. Future work should extend it into a more practical design. Most importantly, a more efficient memory system should be developed, support for floating-point arithmetic should be added, a more extensive software stack such as OpenCL should be implemented, and the design should be evaluated with more lanes on larger FPGAs. Another promising direction is to combine Dolunay's innovations with those of other designs. [10] showed that dynamic scalarization can substantially reduce on-chip memory usage. [14] introduced a novel floorplan solver to support more efficient permutation operations. Finally, [15] implemented a texture unit for Vortex, accelerating 3D applications. These ideas could be combined with ITS capabilities in more advanced GPGPU designs.

References

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," Micro, IEEE, vol. 28, pp. 39 – 55, 04 2008.

[2] J. Nickolls and W. Dally, "The gpu computing era," Micro, IEEE, vol. 30, pp. 56 – 69, 05 2010.

[3] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," Queue , vol. 6, pp. 40–53, 03 2008.

[4] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," Proceedings of the IEEE , vol. 96, pp. 879–899, 05 2008.

[5] NVIDIA Corporation, "Nvidia tesla v100 gpu architecture: The world's most advanced data center gpu," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, accessed: 2026-03-28.

[6] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," 2018. [Online]. Available: https://arxiv.org/abs/1804.06826

[7] F. Elsabbagh, B. Tine, P. Roshan, E. Lyons, E. Kim, D. E. Shim, L. Zhu, S. K. Lim, and H. kim, "Vortex: Opencl compatible risc-v gpgpu," 2020. [Online]. Available: https://arxiv.org/abs/2002.12151

[8] S. Machetti, P. D. Schiavone, L. Orlandic, D. Huang, D. Kasap, G. Ansaloni, and D. Atienza, "e-gpu: An open-source and configurable risc-v graphic processing unit for tinyai applications," 2026. [Online]. Available: https://arxiv.org/abs/2505.08421

[9] C. Collange, "Simty: generalized SIMT execution on RISC-V," in First Workshop on Computer Architecture Research with RISC-V , ser. First Workshop on Computer Architecture Research with RISC-V, vol. 6, Boston, United States, Oct. 2017, p. 6. [Online]. Available: https://inria.hal.science/hal-01622208

[10] M. Naylor, A. Joannou, A. Markettos, P. Metzger, S. Moore, and T. Jones, "Advanced dynamic scalarisation for risc-v gpgpus," 11 2024, pp. 260–267.

[11] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," 01 2008, pp. 407–420.

[12] A. Tino, C. Collange, and A. Seznec, "Simt-x: Extending single-instruction multi-threading to out-of-order cores," ACM Trans. Archit. Code Optim. , vol. 17, no. 2, May 2020. [Online]. Available: https://doi.org/10.1145/3392032

[13] M. Nemirovsky and D. M. Tullsen, Fine-Grain Multithreading . Cham: Springer International Publishing, 2013, pp. 25–31. [Online]. Available: https://doi.org/10.1007/978-3-031-01738-4_4

[14] J. Liu, Q. Li, Y. Luo, H. Zhang, J. Lu, S. Fan, J. Ye, Y. Liu, X. Liu, Y. Yang, Z. Ye, Y. Zeng, A. Shen, R. Huang, W. Cong, X. Zou, and M. Gao, "Titan-i: An open-source, high performance risc-v vector core," in Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture , ser. MICRO '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 675–690. [Online]. Available: https://doi.org/10.1145/3725843.3756059

[15] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, "Vortex: Extending the risc-v isa for gpgpu and 3d-graphics," in MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture , ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 754–766. [Online]. Available: https://doi.org/10.1145/3466752.3480128

Operation Count as a Performance Predictor: An Empirical Study of Lightweight AEAD Schemes

Raphaël Wintersdorff; Renaud Pacalet

COMMUNICATIONS ET ÉLECTRONIQUE [COMELEC]

Télécom Paris
Palaiseau, France

Laurent Sauvage

SECURE AND SAFE HARDWARE [SSH]

Télécom Paris
Palaiseau, France

Abstract—**This paper presents an analytical model for estimating the performance of cryptographic algorithms based on operation counting at the specification level. Aside from the requirement of having 32-bit operations available on the target, the proposed approach avoids platform-dependent assumptions by deriving a cost metric directly from the number of operations required by an algorithm description. This enables early-stage, target-independent performance comparison. The model is evaluated using the NIST lightweight cryptography competition benchmark results, where it is shown to accurately approximate cycle counts across a diverse set of algorithms. The results indicate that operation counting can provide a consistent relative ordering of computational cost, and even a good order of magnitude approximation, despite its simplicity compared to empirical measurement methods. In addition, the model is applied to algorithms protected against side-channel attacks using first-order Boolean masking schemes implemented with ISW gadgets. While the method has not yet been formally validated for masked implementations, it is used as an exploratory indicator of the additional overhead introduced by the masking countermeasure. Overall, this work suggests that specification-level operation counting can serve as a practical and low-cost tool for early performance estimation and comparative analysis of unprotected, and possibly masked cryptographic implementations.**

***Keywords*-performance estimation; lightweight cryptography; boolean masking; software**

I. INTRODUCTION

Performance estimation of cryptographic algorithms is a recurring challenge in both academic research and practical system design. Accurate evaluation is particularly important in the context of lightweight cryptography, where algorithms are expected to operate under strict constraints on computation, memory, and power consumption. Existing evaluation methodologies can broadly be divided into two categories: algorithmlevel analytical evaluation and implementation-based performance estimation. Analytical evaluations of lightweight cryptographic algorithms typically report counts of elementary operations, such as logical, arithmetic, or memory operations for software implementations, or hardware area and Gate Equivalents (GEs) for hardware implementations [7], [18]. While these metrics are useful for comparing algorithmic complexity, they are not generally intended to estimate execution time. Conversely, execution-time estimation is usually performed from an implementation using compiler analyses, instructionset simulators, or empirical benchmarking [6], [17], [20].

In this paper, we propose a performance estimation model that bridges these two levels of evaluation. The model is based on a simple yet systematic principle: counting the elementary operations derived directly from the algorithmic specification and translating these counts into an estimate of execution cost. The underlying premise is that, for a broad class of cryptographic primitives, the structure of the specification provides a sufficiently faithful proxy for computational cost. Unlike compiler-based estimation frameworks, instruction set simulators, or platform-specific benchmarking, the proposed approach requires neither source code nor executable implementations, making it suitable for use at the earliest stages of algorithm development.

We evaluate the validity of this model using the National Institute of Standards and Technology (NIST) lightweight cryptography competition benchmark results. In our analysis, we consider the primary AEAD members of Ascon [11], GIFTCOFB [1], Grain [14], ISAP [10], PHOTON-Beetle [2], Romulus [13], Sparkle [3], TinyJambu [21], and Xoodyak [9]. These algorithms provide a diverse and representative set of modern lightweight primitives, allowing us to compare specificationbased estimates against publicly available benchmark results. Our findings indicate that, despite its simplicity, the proposed model successfully captures the relative computational cost of different algorithms for a given input size and even provides a good approximation of the measured execution time of software implementations.

Beyond unprotected implementations, we further investigate the applicability of the model to implementations protected against side-channel attacks. In particular, we consider first-order Boolean masking schemes implemented using Ishai–

Manuscript received May 11, 2026; revised July 22, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.138

Sahai–Wagner (ISW) gadgets [15]. Such countermeasures introduce substantial computational overhead due to the additional operations and randomness generation, whose impact is generally difficult to assess without a complete implementation-level analysis. Although only a limited number of masked implementations are publicly available for experimental validation, we apply the proposed model as a preliminary indicator of performance for masked implementations.

The contributions of this work are threefold: (i) we propose a specification-level analytical model that estimates implementation performance directly from the algorithm description, without requiring any implementation, (ii) we validate the model against benchmark results from the NIST Lightweight Cryptography competition, demonstrating that it provides meaningful execution-time estimates despite its simplicity, and (iii) we extend the methodology to masked implementations as a preliminary exploratory study. While the proposed model is intentionally simple and is not intended to replace implementation-level benchmarking or cycle-accurate simulation, it provides an early-stage analytical tool for performance assessment, when implementation-based evaluation is not yet possible.

In this paper, we use the following symbols:

$\oplus$	xor operator, exclusive disjunction
$\wedge$	and operator, conjunction
$\neg$	not operator, negation
$A \ggg n$	rot operation, rotation of A n bits to the right
$A \gg n$	shift operation, bit shift of A n bits to the right
$A \ll n$	shift operation, bit shift of A n bits to the left
$A \| B$	concatenation of A and B
$a \leftarrow b$	assignment statement, where variable a takes the value of b

We also use other operators that do not require symbols, namely the inclusive disjunction or, the exclusive disjunction with a constant xor.c, the conjunction with a constant and.c, the modular addition add, and the table lookup lookup.

II. Approximation of performance

In our analysis, we considered 9 algorithms among the 10 finalists of the NIST lightweight cryptography competition. We only removed Elephant [5] from our analysis, since it has a performance dominated by bit permutation operations whose computational cost is highly sensitive to implementation-level optimization strategies. Since the proposed model relies on a static operation-counting abstraction at the specification level, such implementation-dependent effects cannot be consistently and fairly represented. Including this algorithm would therefore introduce a bias unrelated to the intended abstraction level of the analysis.

For every of these algorithms, we counted the number of arithmetic and logic operations that would be performed in the calculation of the outputs, for 32-bit software implementations, based on the specifications given when available. Since some specifications are not naturally amenable to 32-bit operations, we adapted them to only use 32-bit variables by strategically packing some variables in the specification.

We show hereafter an example of operation count on Ascon. While Ascon is natively specified using 64-bit words, its implementation on 32-bit platforms requires an appropriate representation of these state variables. A straightforward approach consists in splitting each 64-bit word into two 32-bit halves, storing the most and least significant bits separately. However, this representation introduces additional complexity for operations involving bit positions from both halves, particularly rotations, which may require multiple instructions and negatively impact performance. To enable more efficient 32-bit implementations, we consider the bit-interleaved representation originally introduced by Bertoni, Daemen, Peeters, and Van Assche for Keccak [4], and later used in reference implementations of Ascon. In this representation, even and odd bits of a 64-bit word are separated and stored in two 32-bit words. This arrangement simplifies the implementation of rotations and other bitwise operations on 32-bit architectures by avoiding costly cross-word manipulations. In this work, bit interleaving is considered as an existing implementation technique for adapting 64-bit cryptographic specifications to narrower architectures. Its use for Ascon is therefore not a contribution of this paper. However, describing this representation is necessary for our operation-count model, since the sequence and number of elementary operations depend on the selected representation, which is not explicitly defined in the Ascon specification.

The 320-bit state S_{BI} of Ascon in its interleaved form can be written as the concatenation of 10 32-bit words:

$$S_{BI} = S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \| S_{0,e} \quad \textbf{(1)}$$

Each round, a constant is added to $S_{2,e}$ and $S_{2,o}$, which costs 2 xor.c.

The bit-interleaved version of the substitution layer can then be done with the following sequence of instructions:

$$\begin{cases} S_{0,*} \leftarrow S_{0,*} \oplus S_{4,*} \\ S_{2,*} \leftarrow S_{2,*} \oplus S_{1,*} \\ S_{4,*} \leftarrow S_{4,*} \oplus S_{3,*} \\ m_{0,*} \leftarrow (\neg S_{0,*}) \wedge S_{1,*} \\ m_{1,*} \leftarrow (\neg S_{1,*}) \wedge S_{2,*} \\ m_{2,*} \leftarrow (\neg S_{2,*}) \wedge S_{3,*} \\ m_{3,*} \leftarrow (\neg S_{3,*}) \wedge S_{4,*} \\ m_{4,*} \leftarrow (\neg S_{4,*}) \wedge S_{0,*} \\ S_{0,*} \leftarrow S_{0,*} \oplus m_{1,*} \\ S_{1,*} \leftarrow S_{1,*} \oplus m_{2,*} \\ S_{2,*} \leftarrow S_{2,*} \oplus m_{3,*} \\ S_{3,*} \leftarrow S_{3,*} \oplus m_{4,*} \\ S_{4,*} \leftarrow S_{4,*} \oplus m_{0,*} \\ S_{1,*} \leftarrow S_{1,*} \oplus S_{0,*} \\ S_{3,*} \leftarrow S_{3,*} \oplus S_{2,*} \\ S_{0,*} \leftarrow S_{0,*} \oplus S_{4,*} \\ S_{2,*} \leftarrow \neg S_{2,*} \end{cases} \quad \textbf{(2)}$$

for $* \in \{e, o\}$.

This substitution layer costs 10 and, 22 xor and 12 not.

Lastly, the linear diffusion layer uses rotations on the words of the state to produce the output. It relies on the use of linear functions $\Sigma_{i,e}$ and $\Sigma_{i,o}$, defined as follows:

$$\begin{aligned} \Sigma_{0,e}(S_{0,e}, S_{0,o}) &= S_{0,e} \oplus ((S_{0,o} \oplus (S_{0,e} \ggg 5)) \ggg 9 \\ \Sigma_{0,o}(S_{0,e}, S_{0,o}) &= S_{0,o} \oplus ((S_{0,e} \oplus (S_{0,o} \ggg 4)) \ggg 1 \\ \Sigma_{1,e}(S_{1,e}, S_{1,o}) &= S_{1,e} \oplus ((S_{1,o} \oplus (S_{1,o} \ggg 11)) \ggg 1 \\ \Sigma_{1,o}(S_{1,e}, S_{1,o}) &= S_{1,o} \oplus ((S_{1,e} \oplus (S_{1,e} \ggg 11)) \ggg 2 \\ \Sigma_{2,e}(S_{2,e}, S_{2,o}) &= S_{2,e} \oplus (S_{2,o} \oplus (S_{2,e} \ggg 3)) \\ \Sigma_{2,o}(S_{2,e}, S_{2,o}) &= S_{2,o} \oplus ((S_{2,e} \oplus (S_{2,o} \ggg 2)) \ggg 1 \\ \Sigma_{3,e}(S_{3,e}, S_{3,o}) &= S_{3,e} \oplus ((S_{3,e} \oplus (S_{3,o} \ggg 3)) \ggg 5 \\ \Sigma_{3,o}(S_{3,e}, S_{3,o}) &= S_{3,o} \oplus ((S_{3,o} \oplus (S_{3,e} \ggg 4)) \ggg 5 \\ \Sigma_{4,e}(S_{4,e}, S_{4,o}) &= S_{4,e} \oplus ((S_{4,o} \oplus (S_{4,o} \ggg 17)) \ggg \\ \Sigma_{4,o}(S_{4,e}, S_{4,o}) &= S_{4,o} \oplus ((S_{4,e} \oplus (S_{4,e} \ggg 17)) \ggg \end{aligned} \quad \textbf{(3)}$$

The state is then updated by using these calculations:

$$(S_{i,e}, S_{i,o}) \leftarrow (\Sigma_{i,e}(S_{i,e}, S_{i,o}), \Sigma_{i,o}(S_{i,e}, S_{i,o})), \quad \textbf{(4)}$$

for $i \in [0\,;4]$.

This linear diffusion layer costs 20 xor and 19 rot.

We thus have a total per-round cost of 10 and, 42 xor, 2 xor.c, 12 not, and 19 rot.

Using the same methodology, we counted the different operations used in the other schemes considered. Table **I** shows the number of operations for each of the main primitives used in the different schemes. There are other functions used in the AEAD modes whose operation count is not included in this paper, such as LFSRs or key schedules for example, but they were taken into account in our final model that we describe hereafter.

In order to perform a comparison between the respective modes of the 9 ciphers considered, we first expressed the number of operations of the modes in terms of the number of operations of the different functions that compose each cipher. For example, if we denote by A_b the number of b-bit words of associated data sent, P_b, the number of b'-bit words of plaintext sent, and $xor_{b''}$ the cost of a b''-bit xor, then the total number of operations of Ascon-AEAD128, the primary member of the Ascon family chosen to become the standard for lightweight cryptography, can be expressed as $A_{128} \cdot (p^b + xor_{128}) + (P_{128} - 1) \cdot p^b + P_{128} \cdot xor_{128} + 2 \cdot p^a + 3 \cdot xor_{128} + xor_1$(Figure 1).

We give the list of number of operations for all the modes considered hereafter:

$$\begin{aligned} C_{\text{Ascon}} = {} & A_{128} \cdot (p^b + xor_{128}) \\ & + (P_{128} - 1) \cdot p^b + P_{128} \cdot xor_{128} \\ & + 2 \cdot p^a + 3 \cdot xor_{128} + xor_1 \\ C_{\text{GIFT}} = {} & A_{128} \cdot (mult2 + G + E_k + 2 \cdot xor_{128}) \\ & + P_{128} \cdot (mult2 + G + E_k + 3 \cdot xor_{128}) \\ & + \text{KeySchedule} \\ C_{\text{Grain}} = {} & (A_{32} + P_{32}) \cdot (2(NFSR + LFSR + y_t) + a_t + r_t) \\ & + \text{Init} \\ C_{\text{ISAP}} = {} & P_{64} \cdot p^6 + (A_{64} + P_{64} + 2) \cdot p^{12} \\ & + (A_{64} + 2P_{64}) \cdot xor_{64} + 4 \cdot p^6 + 254 \cdot p^1 \\ & + 256 \cdot xor_1, \text{where p} = \text{Ascon-p} \\ C_{\text{PHOTON}} = {} & (A_{128} + P_{128} + 1) \cdot f + P_{128} \cdot \rho \\ & + (A_{128} + 2) \cdot xor_{128} \\ C_{\text{Romulus}} = {} & (\lfloor \frac{A_{128}}{2} + P_{128} + 1 \rfloor) \cdot (E_k + \rho) \\ & + \text{TweakeySchedule} \\ C_{\text{Sparkle}} = {} & (A_{256} - 1 + P_{256} - 1) \cdot (S_7 + \rho) \\ & + (A_{256} + P_{256}) \cdot xor_{256} \\ & + 3 \cdot xor_{128} + 2 \cdot S_{11} + 2 \cdot \rho \\ C_{\text{TinyJambu}} = {} & A_{32} \cdot (p_{384} + xor_{32}) \\ & + (P_{32} + 2) \cdot (p_{1024} + 2 \cdot xor_{32}) \\ & + 4 \cdot p_{384} + 6 \cdot xor_3 \\ C_{\text{Xoodyak}} = {} & (A_{352} + P_{192} + 1) \cdot p^{12} + A_{352} \cdot (xor_{352} \\ & + xor_{32}) + 2P_{192} \cdot xor_{192} \end{aligned}$$

Note that the use of the terms $(P_{128} - 1) \cdot p^b$ for Ascon-AEAD128 and $(A_{256} - 1 + P_{256} - 1) \cdot (S_7 + \rho)$ for Sparkle is

Table I: Operation counts for the main primitives

Operation	Bloc ciphers		Permutations						Stream cipher
	GIFT-128 (GIFT-COFB)	Skinny (Romulus)	p^r (Ascon)	p^r (ISAP)	PHOTON$_{256}$ (PHOTON-Beetle)	SPARKLE384 (Sparkle)	P_n (TinyJambu)	XOODOO (Xoodyak)	NFSR (Grain)
and	15	-	10	10	-	-	1	12	14
or	5	-	-	-	-	-	-	-	-
xor	56	44	42	42	56	42	8	36	42
and.c	28	28	-	-	56	-	-	-	-
xor.c	5	3	2	2	8	26	-	1	-
not	5	-	12	12	-	-	1	12	-
shift	28	24	-	-	56	2	8	-	54
rot	-	-	19	19	-	44	-	20	-
add	-	-	-	-	-	24	-	-	-
lookup	-	16	-	-	64	-	-	-	-
Sum	142	115	85	85	240	138	18	81	110
#Rounds	8	40	8/12	1/6/12	12	7/11	12/32	12	-
Total Ops	1136	4600	680/1020	85/510/1020	2880	966/1518	216/576	972	110

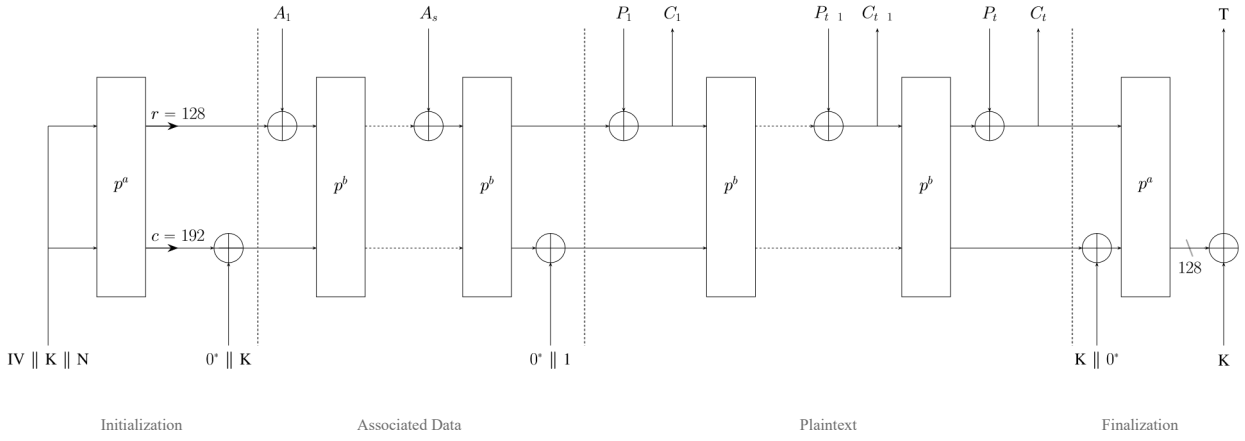


Fig. 1: Graphical representation of the Ascon-AEAD128 mode of operation.

The rate r being equal to 128, associated data and plaintext blocks are treated as 128-bit words. Associated-data absorption contributes A_{128} calls to p^b and A_{128} 128-bit xor, while plaintext processing contributes $P_{128} - 1$ additional calls to p^b and P_{128} 128-bit xor. There are also 2 calls to p^a, in the Initialization and Finalization phases. Finally, there are 3 exclusive or in the bottom branch, with a capacity c 192, but since one of the operands is padded with zeros, this can be considered as a 128-bit xor. Similarly, the addition with $0^* \| 1$ is modeled as a 1-bit xor.

an abuse of notation, used to improve readability: if no plaintext or associated data is provided, terms of the form $P_* - 1$ or $A_* - 1$ should be considered as zero, and not a negative number.

With these expressions, we have access to a platform-agnostic model that can estimate the execution time of the algorithms we considered for any input lengths.

In order to verify the soundness of our method, we made some statistical tests between a list of estimated number of cycles, given by our model for different lengths of associated data and plaintext, and a list of actual number of cycles given by the performance benchmarking results from the NIST's Github page (https://github.com/usnistgov/Lightweight-Cryptography-Benchmarking/tree/main). The results shown in the paper are those from the Arduino Nano 33 BLE (nRF52840) target architecture, but we performed the same experiments on all platforms available for the benchmark. The benchmark consists of cycle counts for different implementations of the finalists of the lightweight cryptography competition organized by the NIST. The cycle counts depend on the number of bytes of associated data and of plaintext. We extracted the cycle counts for byte lengths in the set {0,8,16,32,64,128}, resulting in 36 values per implementation. We selected, for each mode, the minimal cycle count among the implementations of their primary member. For each of the 36 pairs of lengths, we calculated the Kendall rank correlation coefficient, also known

as Kendall's τ [16], which measures how similar the orderings of the implementations are in the sets of modeled and real cycle count. It is calculated as follows:

$$\tau = \frac{\#\text{correctly ordered pairs} - \#\text{incorrectly ordered pairs}}{\#\text{pairs}} \quad (5)$$

τ can be used in a statistical hypothesis test known as τ test, which yielded, in our case, a p-value below 0.01 for every input size. This value is below the usual threshold value of 0.05, allowing us to reject the null hypothesis that the two methods for finding the number of cycles are independent. However, for ease of interpretation, we also converted the results into a percentage of correct pairwise rankings, thanks to the following formula:

$$p = \frac{\tau + 1}{2} \quad (6)$$

We have a mean percentage of correct pairwise ranking of about 0.95, meaning that the model and the measurement agree about 95% of the time on the ranking of any two pairs of algorithm cycle count in the list of considered algorithms (see Figure 2 for the per-size results).

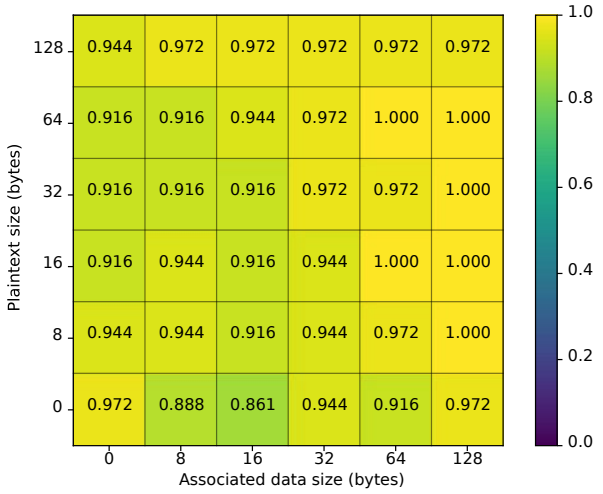


Fig. 2: Percentages of correct pairwise rankings.
Percentages were calculated per input size pair. For a given pair of input sizes, the probability of the method and the benchmark agreeing on the ordering of two algorithms in terms of cycle count is between 0.861 and 1, with an average of 0.951.

While conserving the order is a property that can be beneficial when choosing an algorithm in a list of candidates for a specific usecase, the order of magnitude of the two methods might be vastly different, which is why we also calculated the Pearson's correlation coefficient ρ [19] for every input size. The observed linear correlation across multiple algorithms suggests that the agreement between the proposed model and empirical benchmark results is not limited to a specific algorithm, since we obtained a p-value of less than 10^{-5} for every test. Given this linear relationship between real and modeled time, we made one linear regression per algorithm, on a dataset of points of the form (real cycle count, estimated cycle count), resulting in explained variance ratios ranging from 88\% to nearly 100\% (Table **II**).

Table II: Regression parameters for evaluated algorithms.

Algorithm	Slope	Intercept	R^2
Ascon	0.64232	-7.98728	0.97779
GIFT-COFB	0.81522	-1077.49161	0.98856
Grain	0.90720	-2047.76317	0.99968
ISAP	0.69126	8905.57643	0.88853
PHOTON-Beetle	0.90514	137.59719	0.99995
Romulus	0.81779	-520.36644	0.95275
Sparkle	0.73989	-1105.07928	0.96878
TinyJambu	1.07438	-857.44311	0.95319
Xoodyak	0.57563	612.48003	0.95518

The slopes of the per-algorithm regressions having a similar order of magnitude, we hypothesized that the whole set of points lies close to a single line, which we verified by making a linear regression on all the points of the form (real cycle count, *f*(estimated cycle count), with *f* being the affine transformation that results in a regression line representing the identity function. The resulting regression line has a ratio of explained variance of 99.5%, suggesting that the final model, after the affine transformation, could be a good proxy for cycle count (Figure 3). However, since the variance of the cycle counts of different algorithms could have different orders of magnitude, the ratio of explained variance alone is not sufficient to validate our method.

In order to numerically assess the accuracy of our method, we calculated the orthogonal root mean square error (RMSE) per algorithm, which is the square root of the average squared Euclidean distance between the points and their orthogonal projection on the considered line. The orthogonal RMSE of the different algorithms is relatively low compared to the spread of the evaluated data points, thus indicating that the points lie close to the line representing the identity function (Table **III**).

We tested the same methodology on all the platforms available in the NIST benchmarks. The ratio of explained variance of the resulting regression line is at least 95.3%, except for the Arduino nano every and the Arduino uno, which are the only two platforms based on 8-bit cores.

We thus deem the proposed model suitable for estimating the order of magnitude of the computational cost of algorithms This allows for a time saving when determining which algorithm has better performance in terms of execution speed, since it does not require implementing the algorithms under consideration and does not depend on the choice of a specific target, as long as it is capable of performing 32-bit operations.

However, it is important to point out that not every scheme

Table III: Per-algorithm Orthogonal RMSE and maximum distance of points in the scatter plot.

Algorithm	Orthogonal RMSE	Maximum point distance
Ascon	948	19667
GIFT-COFB	945	39849
Grain	473	95719
ISAP	3702	75756
PHOTON-Beetle	941	282089
Romulus	2201	66516
Sparkle	598	14396
TinyJambu	2370	36336
Xoodyak	1313	16603

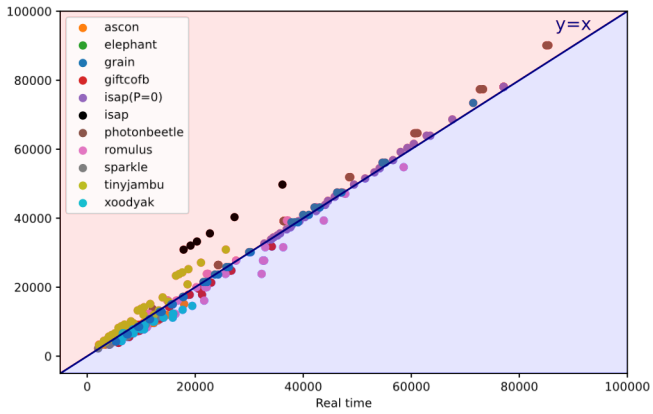


Fig. 3: Scatter plot of model time against real time. Points are mostly aligned along the regression line $y = x$, with a ratio of explained variance of 99.6%, when removing the outliers for ISAP when the plaintext is empty. This line, represented in blue, represents where points would lie if the model had perfect accuracy. The red half-plane indicates where the model suggests worse results than those in the benchmark. Conversely, the blue half-plane indicates where the model suggests better performance than the actual results.

lends itself to such an estimation. For instance, Elephant is the only AEAD scheme from the NIST lightweight cryptography competition that we did not consider in our analysis, due to the fact that the only implementation available in the NIST benchmark is in C and contains a bit permutation that is not optimized. If we were to model this implementation strategy, consisting of selecting each of the 160 bits and accumulating them in different registers, our method would yield a high estimated cost, while in reality the compiled executable runs faster than the model predicts.

It is also important to consider implementation choices that can significantly affect the operation counts. Notably, some algorithms can be implemented with the use of lookup tables, or can benefit from bitslicing. These are design choices that have a substantial impact on performance. Our model is capable of taking such choices into account, thereby enabling a rapid assessment of the potential performance of varied implementation choices.

The algorithms for which our model performs best are those that are primarily defined using arithmetic and logical operators on 32-bit variables, or that can be adapted to operate on 32-bit variables by packing bits or bytes together. Operations that are not straightforward to express in terms of 32-bit variables lead to a mismatch between the modeled cost and the actual cost of the algorithm.

While multiple implementations of the NIST lightweight cryptography competition finalists are available for the different platforms in the performance benchmarking Github repository of the NIST, there are few public implementations of these algorithms protected against side-channel attacks. We surmise that our model is applicable to protected implementations, although this claim has yet to be confirmed via practical experiments. With this assumption in mind, we explore in the rest of the paper the effect of boolean masking on the lightweight modes we considered, despite the lack of publicly available implementation.

III. Effects of Boolean Masking on the Operation Count for the Lightweight Competition Finalists

The method used in Section II for estimating the execution time of unprotected algorithms can be used for protected implementations, though it should be considered as an indicator, until further experiments are done to assess its suitability to masked schemes. Since masking introduces new variables and more computation, it might for instance affect register pressure in assembly implementations, which would in turn make calls to memory more frequent, and since these calls are not taken into account in the method, the difference between real and modeled times could deviate more than for unprotected implementations. Nevertheless, we used it to determine which masked schemes are the most promising without having to implement them all.

The extension to masked implementations is relatively straightforward: instead of having operations between variables accounting for one unit cost, we now have gadgets functionally equivalent to the unmasked operations, which themselves are made of multiple logic operations. We then just have to compute a weighted sum, where the weights are the number of operations for a given gadget (Table **IV**), and the values are the number of the different gadgets used, which is the same as the number of operations used in the unprotected implementations, assuming that no additional randomness in the form of refresh gadgets needs to be added to make the implementation secure. For simplicity, we restrict our analysis on first-order boolean masking. We chose to use the ISW-AND gadget [15] for the logical conjunction and to create the inclusive disjunction, since it allows for an easy generalization to higher-order settings, and it also allows for the creation of formal proofs of security against side-channel attacks in the probing model [15], if required. As for the affine gadgets, we use either the trivial implementation of a gadget when it is linear, and apply the affine part to the first share when it isn't.

TABLE IV: Operation counts for the gadgets considered for first-order ISW masking

Operation	ISW-AND	ISW-OR	XOR	AND.c	XOR.c	SHIFT	ROT	NOT	ADD
and	4	4	-	-	-	-	-	-	40
or	-	-	-	-	-	-	-	-	-
xor	4	4	2	-	-	-	-	-	84
and.c	-	-	-	2	-	-	-	-	-
xor.c	-	-	-	-	1	-	-	-	-
not	-	3	-	-	-	-	-	1	-
shift	-	-	-	-	-	2	-	-	20
rot	-	-	-	-	-	-	2	-	-
add	-	-	-	-	-	-	-	-	-
Sum	8	11	2	2	1	2	2	1	144
#random	1	1	-	-	-	-	-	-	10

Defining an ADD gadget for the arithmetic addition is not as straightforward, since this operation is not linear with respect to the exclusive disjunction operation xor. One way of doing that would be to use conversion algorithms to temporarily switch from boolean masking to arithmetic masking, perform the addition in the arithmetic domain, and then convert the result back to boolean masking [12]. It is also possible to avoid this back and forth by using a masked implementation of a full binary adder with the previously defined gadgets. This adder can be a ripple-carry adder, or a carry-lookahead adder such as the Kogge–Stone adder, for example [8]. However, both methods are costly in terms of both computational complexity and randomness consumption, which makes them unattractive when arithmetic additions occur frequently, as is the case in ARX-based permutations such as SPARKLE384[11]. In the rest of the paper, we consider the addition gadget to be a Kogge-Stone masked addition algorithm [8].

As for lookup tables, the conversion to a masked lookup table is less easy: for 2-sharings, if a table has 256 possible indices for instance, then its masked version would require $256^2 = 65536$ entries. The amount of computation and memory to use this kind of table would be prohibitive for lightweight applications, and scales poorly with the masking order. A solution is to fix the masks of the varibales that are used as inputs to the table, and add another mask at the output. More formally, for an input mask value $m \in E$, an output mask value $m' \in E$ and a table $T: E \to E$, we define a table $T_m: E \to E$ such that, for all $i \in E$, $T_m[i] = T[m \oplus i\,] \oplus m'$. This ensures that for a 2-sharing $[x] = (x_m, m)$, we have $T_m[x_m] = T[m \oplus x_m] \oplus m' = T[x] \oplus m'$, which is a masked value of the *secret* $T[x]$ that we aim to calculate. The use of such masked tables introduces new problems, however: since their definition depends on the values m and m′, they need to be recomputed each time the masks change. Since the reuse of mask is not advised for security purposes, the tables would need to be calculated for each encryption, negating the advantage of using them for fast calculations.

We removed the implementations that made use of tables, namely those of PHOTON-Beetle and Romulus, for which a tabulated S-box and other precomputations were used. After removal, we made regressions on the modeled costs of masked implementation. With the exception of Sparkle, whose use of arithmetic additions makes the cost overhead very large, the cost of the considered masked implementations ranges from about 2.7 to 3.7 times that of their unprotected counterpart, according to our model (Figure 4).

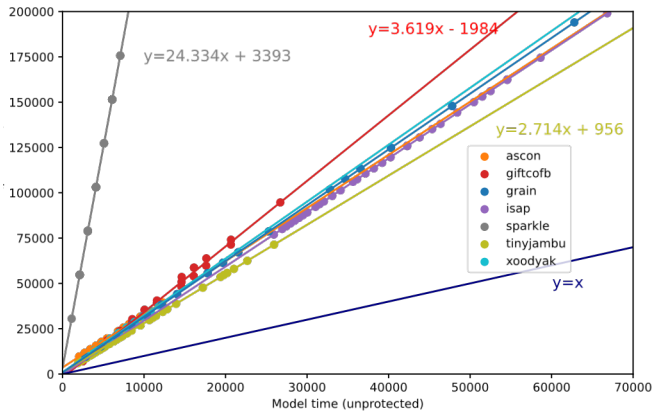


Fig. 4: Scatter plot of model time for masked implementation against model time for unprotected implementation, with removed table-based implementations. Except for Sparkle, regression lines for the different algorithms have a slope between 2.7 and 3.7. These similar values suggest that their ranking for any input size pair is

mostly unchanged compared to the ranking of the unprotected algorithms.

This indicates that for any given pair of input size, the ranking of these algorithms would not change much, except for Sparkle, which becomes quickly unattractive when the input sizes get larger, since a regression on the data points related to Sparkle yields a line $y = 24.334x + 3393$, indicating that a masked implementation for this algorithm would be approximately 24 times slower than the unprotected one.

IV. Conclusion

This paper presented a model for estimating the performance of cryptographic algorithms through operation counting derived directly from their specification. The proposed approach aims to provide a simple method for evaluating computational cost during early design and analysis stages, without relying on platform-specific benchmarking. The relevance of the model was investigated using algorithms from the NIST lightweight cryptography competition benchmark results. These include a wide range of cryptographic designs, such as block ciphers, permutation-based ciphers, and a stream cipher. Our results show that specification-level operation counting provides a useful indicator of algorithmic complexity across different lightweight cryptographic constructions.

In addition, the method was applied to implementations protected against side-channel attacks using first-order Boolean mmasking based on ISW gadgets. Although the model has not yet been experimentally validated for masked implementations, the obtained estimates suggest that it can still provide useful insight into the computational overhead introduced by masking countermeasures. This preliminary study highlights the potential of operation-based estimation techniques for comparing secure implementations at a high level of abstraction.

The simplicity of the proposed model constitutes both its main strength and its primary limitation. While it enables fast and portable evaluation, it does not capture target-dependent effects that could make the model results deviate from actual cycle counts. Future work could focus on refining the model, to take into account access to memory for instance. Another direction would be to extend the validation to additional implementation styles, algorithms, and masking strategies, and to investigate weighting strategies for different operation classes in order to improve estimation accuracy. Since our methodology does not necessarily need the operations to be between 32-bit variables, future work could be about adapting this method for 8-bit variables and verifying its applicability through benchmarks results on 8-bit platforms.

Overall, this work demonstrates that specification-level operation counting can serve as a practical and low-cost tool for performance analysis of lightweight cryptographic algorithms, and shows how it can be used to estimate the cost of their protected counterparts.

REFERENCES

[1] Subhadeep Banik, Avik Chakraborti, Akiko Inoue, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT-COFB. Cryptology ePrint Archive, Paper 2020/738, 2020.

[2] Zhenzhen Bao, Avik Chakraborti, Nilanjan Datta, Jian Guo, Mridul Nandi, Thomas Peyrin, and Kan Yasuda. Photon-beetle, 2021. Accessed: 2024-12-11.

[3] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschadl, Amir Moradi, L¨ eo Perrin, Aein Rezaei Shahmirzadi, Aleksei´ Udovenko, Vesselin Velichkov, and Qingju Wang. Schwaemm and esch: Lightweight authenticated encryption and hashing using the sparkle permutation family, 2021. Accessed: 2024-12-11.

[4] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche.¨ Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 313–314, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] Tim Beyne, Yu Long Chen, Christoph Dobraunig, and Bart Mennink. Elephant v2, 2021. Accessed: 2024-12-11.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[7] Alex Biryukov and Leo Perrin. State of the art in lightweight symmetric´ cryptography. *IACR Cryptol. ePrint Arch.*, 2017:511, 2017.

[8] Jean-Sebastien Coron, Johann Großsch´ adl, and Praveen Vadnala. Secure¨ conversion between boolean and arithmetic masking of any order. pages 188–205, 09 2014.

[9] Joan Daemen, Seth Hoffert, Silvia Mella, Michael Peeters, Gilles Van¨ Assche, and Ronny Van Keer. Xoodyak, a lightweight cryptographic scheme, 2021. Accessed: 2024-12-11.

[10] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. Isapv2.0, 2021. Accessed: 2024-12-11.

[11] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schlaffer. Ascon v1.2: Lightweight authenticated encryption and hash-¨ ing. *J. Cryptol.*, 34(3), July 2021.

[12] Louis Goubin. A sound method for switching between boolean and arithmetic masking. pages 3–15, 05 2001.

[13] Chun Guo, Tetsu Iwata, Mustafa Khairallah nad Kazuhiko Minematsu, and Thomas Peyrin. Romulus-v1.3, 2021. Accessed: 2024-12-11.

[14] Martin Hell, Thomas Johansson, Alexander Maximov, Willi Meier, Jonathan Sonnerup, and Hirotaka Yoshida.¨ Grain-128aeadv2, 2021. Accessed: 2024-12-11.

[15] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 463–481, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[16] M. G. Kendall. Rank correlation methods. 1949.

[17] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[18] Kerry McKay, Lawrence Bassham, Meltem Sonmez Turan, and Nicky Mouha. Report on lightweight cryptography, 2017-03-28 00:03:00 2017.

[19] Karl Pearson and Francis Galton. Vii. note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58(347-352):240–242, 1895.

[20] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstrom. The worst-case execution-¨ time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008.

[21] Hongjun Wu and Tao Huang. Tinyjambu: A family of lightweight authenticated encryption algorithms(version 2), 2021. Accessed: 202412-11.

Instruction Flow Amplifier: A Novel Architecture for In-Order Processor Front-End Optimization

João Antônio Temochko Andre
Department of Electrical Engineering
Centro Universitário FEI
São Bernardo do Campo, Brazil
joao.temochko@ifsp.edu.br

Alexandre Beletti Ferreira
Department of Informatics
Federal Institute of Sao Paulo
Sao Paulo, Brazil
higuita@ifsp.edu.br

***Abstract*—This paper proposes the Instruction Flow Amplifier (IFA), a novel architecture designed to mitigate front-end bottlenecks in in-order processor cores. Using the Gem5 simulator, we evaluate Cycles Per Instruction (CPI), Instructions Per Cycle (IPC), branch prediction accuracy, and instruction cache misses to demonstrate front-end bottlenecks and quantify the performance gains achieved by the proposed architecture. Unlike traditional trace caches that primarily focus on instruction storage, the proposed IFA integrates a lightweight Dependency Analysis Unit (DAU) to enable pseudo-out-of-order dispatch within an energy-efficient in-order infrastructure. The experimental results demonstrate that the IFA acts as an effective instruction supply accelerator. In workloads dominated by tight loops (MatMult), the proposed architecture effectively saturates the pipeline, achieving a 28% IPC improvement. In control-heavy workloads (WikiSort), the IFA mitigates fetch latency through aggressive buffering, improving IPC by 13.4% with negligible impact on branch prediction accuracy.**

***Index Terms*—Computer Architecture, RISC-V, ARM, Front-End, Gem5, In-Order Processor cores.**

I. Introduction

Instruction fetch remains a fundamental performance bottleneck in modern processors [1]. In-order processor cores are particularly vulnerable to stalls, which completely halt execution when dependent instructions are encountered [2], [3]. Additionally, branch mispredictions and control dependencies fragment the instruction stream [2], while data dependencies serialize execution despite the presence of instruction-level parallelism.

Contemporary processor cores address these limitations through larger caches and wider fetch mechanisms, increasing complexity and power consumption without proportional performance gains [1], [4].

The performance of a processor, regardless of the efficiency of its execution core (back-end), is fundamentally limited by the rate at which useful instructions can be supplied. The instruction fetch subsystem, or front-end, is responsible for supplying the pipeline with a continuous stream of instructions while anticipating program control flow to minimize functional-unit idle time [1], [5].

Main memory access latency remains one of the primary obstacles to modern processor performance. In the front-end context, an instruction cache miss (L1-I miss) can completely stall the processor pipeline, exposing the core to hundreds of idle cycles [6]. To mitigate this effect, speculative prefetching techniques have become ubiquitous, aiming to load instruction blocks into the cache before they are required by the execution flow [1], [7], [8].

One of the most prominent approaches for mitigating memory-access latency is the *Fetch Target Queue (FTQ)*, originally proposed by Reinman et al. [5]. Instead of coupling branch prediction directly to instruction-cache access, the FTQ architecture allows the branch predictor to run ahead, generating target addresses and storing them in a queue.

In addition to the FTQ, which stores target addresses, advanced architectures employ *Instruction Byte Buffers (IBBs)* to temporarily store fetched instructions prior to decoding. In designs such as the Decoded Instruction Architecture (DIA) proposed by Santana et al., this concept is extended to store pre-decoded instructions or micro-operations, thereby removing decoding latency from the critical execution path [9].

Despite these latency benefits, most decoupled architectures, including FTQ-based designs, manage their buffers as strict FIFO (First-In, First-Out) queues. This introduces a critical limitation known as head-of-line blocking.

As characterized by Chacon and Gino, when the instruction at the head of the queue stalls due to a cache miss (Stalling Head Instruction), the entire instruction stream is interrupted, even if valid instructions are available in later queue positions [8]. In fully *out-of-order (OoO)* processors, this issue is mitigated through large instruction windows and register renaming. However, in in-order or embedded processors, where such structures are prohibitively expensive, the FIFO model renders decoupling ineffective under long-latency data dependencies.

This paper proposes the *Instruction Flow Amplifier (IFA)*, a lightweight front-end component that implements real-time register dependency analysis. Instead of relying on conservative scheduling policies, the IFA identifies independent instruction sets within an instruction window, enabling more efficient dispatch. The proposed design is inspired by classical instruction scheduling theory and out-of-order execution principles, adapted for practical single-core implementations.

The proposed architecture aims to answer the following research questions (RQs):

Manuscript received May 11, 2026; revised May 26, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.139

- RQ1: Can the IPC of in-order processors be increased through front-end optimization?
- RQ2: Which additional performance parameters can be improved by the proposed solution?
- RQ3: Does the proposed solution incur lower energy overhead compared to strategies?

The remainder of this paper is organized as follows: Section II presents the related work, Section III presents the architectural concept, Section IV presents the performance analysis, Section V discusses the performance and speedup metrics, Section VI evaluates the energy efficiency, and Section VII concludes the paper.

II. Related Works

A. Fetch Target Queue (FTQ)

Originally proposed by Reinman et al. [5], the FTQ serves as an interface between branch prediction and the memory subsystem. Because the bandwidth and operating speed of the branch predictor, which manipulates only target addresses, are typically higher than those of the instruction cache, the FTQ enables deeper control-flow lookahead. Previous studies have shown that FTQ-based architectures can better tolerate instruction cache misses (L1-I misses), since the queue maintains pending fetch requests ready for processing as soon as memory access becomes available [8], [10].

B. Instruction Byte Buffer (IBB) and Pre-Decoding

In addition to the FTQ, which stores target addresses, advanced architectures employ Instruction Byte Buffers (IBBs) to temporarily hold fetched instructions prior to decoding.

These structures allow in-order processors to maintain high functional-unit utilization, provided that the instruction buffer does not become empty. Oberoi argues that a truly high-performance front-end should be capable of fetching instruction fragments out of program order to maximize memory bandwidth utilization, although sequential reassembly is typically required before dispatch in conventional architectures [11].

C. Latency Tolerance: Runahead and iCFP

One approach for mitigating the impact of long-latency cache misses (L2/L3 misses) is runahead execution. Originally proposed as a mechanism to reduce the complexity of large instruction windows, this technique allows the processor to continue speculative execution after a blocking event, such as a load miss. During this phase, data dependencies are temporarily ignored to generate prefetches for future instructions and data. At the end of the blocking period, the speculative state is discarded and normal execution resumes [12]. Although effective at tolerating memory latency, runahead execution wastes energy by executing instructions whose results are eventually discarded.

D. Front-end Execution Architecture (FXA)

Shioya et al. proposed a distinct architectural approach known as the Front-end Execution Architecture (FXA). FXA introduces a small and energy-efficient in-order execution unit (IXU) immediately after the fetch stage, preceding a conventional out-of-order execution unit (OXU) [13].

The premise of FXA is that a significant portion of instructions (more than 50%) can be executed immediately in order if their operands are ready, thereby avoiding the complex logic associated with dynamic scheduling. Instructions that cannot be executed in the IXU are forwarded as NOPs to the OXU.

Although FXA improves the energy efficiency of high-performance processors by reducing pressure on OoO scheduling logic, it still depends on the presence of a complex back-end to handle unresolved dependencies and speculative execution scenarios. This differs from the proposed IFA architecture, which aims to optimize a purely in-order core [13].

E. The difference between IFA and other Architectures

Unlike the previously discussed approaches, the Instruction Flow Amplifier (IFA) transforms the decoupling buffer into an inspectable dispatch structure rather than a strict FIFO queue. This design enables the mitigation of head-of-line blocking without incurring the complexity and energy cost of a fully out-of-order core, addressing a gap identified in the literature on decoupled front-end architectures.

The IFA operates strictly within a bounded front-end instruction window. Unlike iCFP, the proposed architecture does not require complex re-execution buffers, and unlike FXA, it does not depend on a secondary out-of-order execution engine. Instead, the IFA exploits the visibility provided by the buffered instruction window to perform pseudo-out-of-order dispatch while locally resolving RAW (Read-After-Write) hazards, thereby maintaining high functional-unit utilization.

III. Architectural Concept

The proposed Instruction Flow Amplifier (IFA) acts as an intelligent front-end scheduler positioned between the fetch and decode stages of an in-order processor core. Unlike traditional trace caches, which are primarily designed to store execution paths, the IFA employs a dynamic instruction window to identify and schedule independent instructions, thereby enabling pseudo-out-of-order execution behavior within an energy-efficient in-order pipeline.

The proposed architecture consists of four main components: the Front-End Buffer (FEB), the Dependency Analysis Unit (DAU), the Dispatcher Unit (DU), and the Dynamic Barrier Synchronization (DBS) mechanism.

A. Front-End Buffer (FEB)

The Front-End Buffer (FEB) serves as a decoupling structure that mitigates latency between the instruction fetch stage and the execution back-end. Functionally, the FEB operates as a double-ended queue (deque) rather than a strictly ordered FIFO structure.

The buffer maintains a sliding window of fetched instructions, typically ranging from 16 to 32 entries. By buffering multiple instructions, the IFA exposes a deeper lookahead window to the control logic, enabling the hardware to analyze instructions beyond the head of the queue.

This structure absorbs front-end stalls caused by cache misses, ensuring a continuous supply of candidate instructions for dispatch.

B. Dependency Analysis Unit (DAU)

The Dependency Analysis Unit (DAU) is the central control structure responsible for ensuring data dependency correctness. It dynamically scans the instructions residing in the FEB to detect Read-After-Write (RAW) hazards.

At every clock cycle, the DAU performs the following operations:

- **Operand Comparison:** The DAU compares the source registers of younger instructions against the destination registers of older, unexecuted instructions within the instruction window.
- **Status Tagging:** Each instruction in the buffer is assigned a status flag (e.g., Ready or Stalled). An instruction is marked as Stalled if it depends on a result that has not yet been produced by the execution units.
- **Non-Blocking Selection:** By identifying independent instructions, the DAU enables the dispatcher to bypass stalled instructions and issue younger ready-to-execute operations, thereby hiding execution latency.

C. Dispatcher Unit (DU)

The Dispatcher Unit (DU) selects instructions from the FEB and forwards them to the execution pipeline. To preserve architectural correctness without requiring complex speculative recovery mechanisms, the DU employs a set of lightweight control barriers.

- **Selection Logic:** The dispatcher scans the instruction window for operations marked as Ready by the DAU and issues them according to the superscalar width supported by the processor core.
- **Control Barriers:** When a branch or jump instruction is encountered, the dispatcher imposes a soft synchronization barrier. Instructions located after the unresolved control-flow operation are temporarily prevented from being dispatched until the branch outcome is resolved. This mechanism avoids unsafe speculative execution while still allowing independent instructions preceding the branch to be reordered safely.

D. Dynamic Barrier Synchronization (DBS)

To preserve architectural correctness while avoiding the latency overhead associated with multi-phase control-state machines, the IFA employs a combinational control mechanism referred to as Dynamic Barrier Synchronization (DBS).

Instead of transitioning between explicit "performance" and "drain" operating modes, the DAU and DU cooperate dynamically to enforce strict instruction ordering only when required. Whenever a critical instruction is detected within the instruction window, such as a system call, atomic operation, or unresolved branch, the control logic immediately establishes a soft synchronization barrier.

This barrier prevents younger instructions from bypassing the critical operation. Consequently, pseudo-out-of-order dispatch is temporarily suspended, and older instructions are drained sequentially until the critical instruction reaches the head of the queue and is safely dispatched.

Once the critical operation is committed to the execution back-end, the barrier is automatically removed, restoring full dispatch capability without introducing additional pipeline state transitions or recovery latency.

IV. Performance Analysis

To evaluate the performance gains provided by the proposed IFA architecture, we used the Gem5 simulator [14] and benchmark suites from MiBench [15] and Embench to compare an in-order processor core (MinorCPU) operating with and without the IFA extension.

The evaluation considers Cycles Per Instruction (CPI), Instructions Per Cycle (IPC), instruction cache misses, branch prediction accuracy, and pipeline stall behavior.

The experimental results demonstrate measurable performance improvements across all evaluated workloads. Performance gains ranged from approximately 3% to 8% for MiBench workloads and reached up to 28% for selected Embench kernels. These improvements reduced execution stalls and increased effective pipeline utilization, contributing to higher overall energy efficiency.

A. The Gem5 Configuration

In the Gem5 simulator, we modified the MinorCPU implementation to emulate an in-order processor enhanced with the proposed IFA mechanism. The primary architectural modification was introduced in the Fetch2 stage. This design decision was motivated by three architectural constraints:

- **Availability of Semantic Information:** The DAU requires access to source and destination register identifiers to detect Read-After-Write (RAW) hazards. In the baseline MinorCPU architecture, Fetch1 performs raw cacheline retrieval, while instruction decoding occurs in Fetch2. Therefore, Fetch2 represents the earliest pipeline stage at which semantic information required by dependency analysis becomes available.
- **Front-End Decoupling:** The IFA Buffer acts as an instruction reservoir capable of absorbing latency bubbles caused by instruction cache misses. By intercepting decoded instructions immediately after Fetch2 and before dispatch, the IFA accumulates a pool of executable instructions, allowing the execution back-end to continue operation even when Fetch1 stalls waiting for memory.
- **Control-Flow Resolution:** Branch prediction validation and target calculation occur within Fetch2. Since the IFA implements control barriers to avoid unsafe speculative

dispatch, it must remain tightly coupled with branch-resolution logic to correctly halt and resume instruction issue based on control-flow resolution.

Consequently, the Fetch2 stage was modified to redirect decoded instructions into the FEB instead of forwarding them directly to the decode/dispatch queue, effectively transforming the baseline pipeline into a decoupled fetch-execute architecture.

The central implementation changes include modifications to the `hasDependency()` function, which detects register dependencies, and to the `evaluate()` function, which controls IFA activation and emulates the scheduling differences between the baseline and enhanced architectures.

B. Workload

Following methodologies adopted in prior work, we selected benchmark applications from MiBench and Embench to evaluate the proposed architecture. The workloads were divided into three categories: mathematical computation, validation, and sorting.

Mathematical Computation: We used BasicMath from MiBench, which performs computationally intensive mathematical operations such as cubic-function evaluation, integer square-root calculation, and angle conversion. We also evaluated MatMult from Embench, which performs dense matrix multiplication over non-zero matrix elements.

Validation: This category includes Blowfish from MiBench and CRC32 from Embench. Blowfish performs symmetric encryption and decryption using logical operations such as XOR and AND, while CRC32 computes cyclic redundancy checks using XOR and bit-shift operations.

Sorting: We evaluated WikiSort from Embench, which implements block merge sorting designed to operate efficiently without requiring significant auxiliary memory allocation.

C. Results

The experimental results demonstrate the effectiveness of the proposed IFA architecture across diverse workloads and instruction set architectures.

As shown in Tables I to X, performance improvements were consistently observed regardless of the target ISA. CPI, stall counts, and instruction-cache misses generally decreased, while IPC increased. These gains were achieved solely through front-end modifications, without altering execution back-end structures.

For example, in the BasicMath benchmark(Tables I and II), CPI was reduced by 7.53% in the RISC-V configuration, while IPC increased by 8.14%. Similarly, Blowfish exhibited a CPI reduction of 4.65% and an IPC improvement of 4.88%.

The most significant improvements were observed in newer Embench workloads optimized for embedded systems. These benchmarks highlight the effectiveness of the IFA under highly structured instruction-level parallelism.

In the MatMult benchmark (Tables III and IV), CPI was reduced by 22.3% in RISC-V and 13.57% in ARM while IPC increased by 28.8% in the RISC-V ISA and 15.70% in the ARM ISA.

TABLE I
RISC-V PERFORMANCE ANALYSIS (BASICMATH)

Metric	Base	IFA	Diff (%)
CPI	1.68	1.56	**-7.53%**
IPC	0.59	0.64	**+8.14%**
Stalls	66.5M	55.4M	**-16.73%**
ICache Miss	4.02M	3.81M	-5.23%
Branch Acc.	94.6%	93.5%	-1.21%
Mispredicts	0.95M	1.24M	+30.43%

TABLE II
ARM AARCH64 PERFORMANCE ANALYSIS (BASICMATH)

Metric	Base	IFA	Diff (%)
CPI	1.99	1.84	**-7.78%**
IPC	0.50	0.54	**+8.44%**
Stalls	66.6M	58.3M	**-12.44%**
ICache Miss	4.27M	4.09M	-4.29%
Branch Acc.	93.5%	93.9%	+0.36%
Mispredicts	0.90M	0.90M	+0.01%

In CRC32 (Tables V and VI), CPI decreased by 17.82% in RISC-V and 18.14% in ARM, IPC improved by 21.68% in RISC-V and 22.16% in ARM.

WikiSort (Tables VII and VIII) also demonstrated substantial gains, with CPI reduced by 11.84% in RISC-V and 13.57% in ARM, IPC increasing by 13.43% in RISC-V and 15.70% in ARM.

The Blowfish benchmark (Tables IX and X) showed smaller gains, the CPI reduced 6.01% in RISC-V and 4.65% in ARM, the IPC gains is 6.39% in RISC-V and 4.88% in ARM.

This tests demonstrated that IFA achieves substantial gains in benchmarks on both RISC-V and ARM ISAs, showing that the gains are ISA-independent. The results reveal that IFA excels at matrix multiplication but yields minimal gains in cryptography algorithms. This illustrates precisely how IFA boosts performance when benchmarks show lower instruction dependency, yet fails to improve when tests exhibit high dependency, is the case with Blowfish or BasicMath.

The results indicate that the IFA effectively mitigates front-end stalls, allowing the processor pipeline to sustain higher throughput while preserving architectural correctness. The improvements are particularly pronounced in workloads exhibiting predictable dependency structures and moderate branch-control complexity.

V. PERFORMANCE AND SPEEDUP METRICS

To quantify the overall performance gains achieved by the proposed architecture, we analyze the observed improvements using Amdahl's Law [16], [17].

The overall speedup of a processor is determined by the fraction of execution time affected by a given optimization and by the magnitude of the acceleration applied to that portion of the system. Since the IFA exclusively optimizes front-end behavior, the maximum achievable speedup remains bounded

TABLE III
RISC-V PERFORMANCE ANALYSIS (MATMULT)

Metric	Base	IFA	Diff (%)
CPI	1.33	1.03	**-22.3%**
IPC	0.75	0.97	**+28.8%**
Stalls	66.6M	58.3M	**-12.44%**
ICache Miss	448	449	0.00%
Branch Acc.	95,10%	95.32%	+0.22%
Mispredicts	1.95M	1.94M	-0.01%

TABLE IV
ARM AARCH64 PERFORMANCE ANALYSIS (MATMULT)

Metric	Base	IFA	Diff (%)
CPI	1.40	1.21	**-13.57%**
IPC	0.72	0.83	**+15.70%**
Stalls	137.42M	71.68M	**-47.84%**
ICache Miss	546	546	0.00%
Branch Acc.	95.04%	95.04%	0.00%
Mispredicts	1.95M	1.95M	0.00%

TABLE V
RISC-V PERFORMANCE ANALYSIS (CRC32)

Metric	Base	IFA	Diff (%)
CPI	1.22	1.00	**-17.82%**
IPC	0.82	1.00	**+21.68%**
Stalls	87.56M	579.32K	**-99.34%**
ICache Miss	411	420	+2.19%
Branch Acc.	99.97%	99.97%	0.00%
Mispredicts	17.52K	17.52K	-0.03%

TABLE VI
ARM AARCH64 PERFORMANCE ANALYSIS (CRC32)

Metric	Base	IFA	Diff (%)
CPI	1.30	1.06	**-18.14%**
IPC	0.77	0.94	**+22.16%**
Stalls	87.58M	17.97M	**-79.48%**
ICache Miss	522	528	+1.15%
Branch Acc.	99.97%	99.97%	0.00%
Mispredicts	17.56K	17.55K	-0.03%

TABLE VII
RISC-V PERFORMANCE ANALYSIS (WIKISORT)

Metric	Base	IFA	Diff (%)
CPI	1.24	1.09	**-11.84%**
IPC	0.81	0.91	**+13.43%**
Stalls	8.73M	3.40M	**-61.05%**
ICache Miss	505	516	+2.18%
Branch Acc.	96.81%	96.85%	+0.05%
Mispredicts	229.18K	225.94K	-1.41%

by the fraction of execution time spent stalled in the instruction supply pipeline.

The speedup model is expressed as follows:

$$S_{\text{speedup}} = \frac{1}{(1 - f_{\text{FE}}) + \frac{f_{\text{FE}}}{S_{\text{IFA}}}} \tag{1}$$

TABLE VIII
ARM AARCH64 PERFORMANCE ANALYSIS (WIKISORT)

Metric	Base	IFA	Diff (%)
CPI	1.36	1.17	**-13.57%**
IPC	0.74	0.85	**+15.70%**
Stalls	12.74M	6.18M	**-51.51%**
ICache Miss	636	643	+1.10%
Branch Acc.	96.34%	96.34%	0.00%
Mispredicts	229.65K	229.75K	+0.04%

TABLE IX
RISC-V PERFORMANCE ANALYSIS (BLOWFISH)

Metric	Base	IFA	Diff (%)
CPI	2.81	2.64	**-6.01%**
IPC	0.36	0.38	**+6.39%**
Stalls	210.60K	190.97K	**-9.32%**
ICache Miss	505	509	+0.79%
Branch Acc.	98.03%	98.05%	+0.02%
Mispredicts	579	573	-1.04%

TABLE X
ARM AARCH64 PERFORMANCE ANALYSIS (BLOWFISH)

Metric	Base	IFA	Diff (%)
CPI	3.01	2.87	**-4.65%**
IPC	0.33	0.35	**+4.88%**
Stalls	180.73K	168.16K	**-6.96%**
ICache Miss	592	586	-1.01%
Branch Acc.	98.05%	98.05%	0.00%
Mispredicts	589	589	0.00%

where:

- f_{FE} represents the fraction of execution time limited by front-end stalls;
- S_{IFA} represents the acceleration factor provided by the IFA for front-end operations.

To contextualize the measured improvements, we apply Amdahl's formulation to the empirical CPI reductions obtained during simulation.

The observed overall speedup is computed as:

$$\text{Speedup} = \frac{T_{\text{baseline}}}{T_{\text{IFA}}} = \frac{1}{1 - \text{TimeReduction}} \approx 1.081 \tag{2}$$

Using the BasicMath benchmark in the RISC-V configuration, the IFA reduced CPI by approximately 7.53%, corresponding to an overall speedup of nearly 1.08.

This result suggests that front-end stalls accounted for approximately 7.5% of the total execution time, validating the initial bottleneck hypothesis predicted by Amdahl's Law.

The results further demonstrate that relatively small front-end optimizations can produce measurable system-wide performance improvements, even without modifications to execution back-end structures such as ALUs, reorder buffers, or cache hierarchies.

Moreover, the measured gains indicate that the instruction supply subsystem remains a critical limiting factor in in-

order architectures, particularly in workloads exhibiting high instruction-level parallelism and frequent front-end stalls.

VI. Energy Efficiency

To validate the Energy Efficiency we use the concept of *Race to Halt* [16] to prove the IFA is better to FIFO (First In, First Out) front-end, many found in in-order processor's [3], [18].

We increased the processor's total power by adding the IFA. Clearly, FEB, DAU, and DU will consume more energy when the instruction comes from memory. However, with IPC gains and a decrease in CPI, stalls, and ICache misses, we managed to make processing faster, thus reducing it by approximately 6.5% for RISC-V and 7.5% for ARM (using metrics from the two benchmarks used).

To rigorously evaluate the energy benefits of the Instruction Flow Amplifier (IFA), we adopt a conservative "Race-to-Halt" model. We posit that the addition of the IFA Buffer and Dependency Analysis Unit (DAU) introduces a power overhead due to the extra transistor count and switching activity of the comparators.

"Previous implementations of front-end enhancements, such as Execution Drafting and the Front-end Execution Architecture (FXA), have demonstrated area overheads ranging from approximately 1% to 2.7% of the core area [13], [19].

Based on literature regarding lightweight out-of-order extensions for in-order cores which reports a 2.4% [20] area overhead for similar issue logic. We model a pessimistic scenario where the IFA increases the core's active power consumption (P_{active}) by **3.5%**.

This approach aligns with the 'Race-to-Halt' strategy described by Miyoshi et al. [21], where execution time reduction is prioritized to maximize the duration of low-leakage sleep states.

The net energy savings (ΔE) are calculated by comparing the baseline energy against [16] the IFA energy, which benefits from reduced execution time (T_{exec}) but suffers from the power penalty ($P_{\text{penalty}} = 1.035 \times P_{\text{base}}$):

$$\Delta E = \left(1 - \frac{(1.035 \times T_{\text{IFA}}) + (P_{\text{idle}} \times T_{\text{saved}})}{1.0 \times T_{\text{base}}}\right) \times 100 \quad (3)$$

1) Results Discussion: Using the empirical data from the BasicMath benchmark (RISC-V), where the IFA reduced execution time by **7.53%** ($T_{\text{IFA}} \approx 0.925 T_{\text{base}}$):

$$E_{\text{IFA}} \approx (1.035 \times 0.925) + (0.1 \times 0.075) \approx 0.965 \quad (4)$$

$$\Delta E \approx 1.0 - 0.965 = \mathbf{3.5}\% \quad (5)$$

Despite the modeled 3.5% power overhead, the system achieves a **3.5% net reduction in total energy**. This confirms that the performance gains from mitigating front-end stalls outweigh the cost of the additional logic, validating the IFA as an energy-efficient architectural enhancement.

This result demonstrates that the performance speedup provided by the IFA is sufficient to offset its logic overhead. By allowing the core to enter the sleep state earlier, the architecture effectively converts microarchitectural throughput gains into tangible battery life extension, validating the Race-to-Halt approach for this design [22], [23].

VII. Conclusion

This paper presented the Instruction Flow Amplifier (IFA), a lightweight front-end architecture that introduces pseudo-out-of-order dispatch capabilities into purely in-order processor cores without requiring modifications to the execution back-end.

By combining a Front-End Buffer (FEB), a real-time Dependency Analysis Unit (DAU), and a Dynamic Barrier Synchronization (DBS) mechanism, the proposed architecture mitigates the head-of-line blocking problem commonly found in FIFO-based decoupled front-end designs.

RQ1 — Can the IPC of in-order processors be increased through front-end optimization? The experimental results demonstrate that it is possible to improve the IPC of in-order processors exclusively through front-end optimization. Across all evaluated workloads and target ISAs (RISC-V and ARM AArch64), the IFA consistently improved IPC while reducing CPI and pipeline stalls. The observed IPC gains ranged from 4.88% in Blowfish (ARM AArch64) to 28.8% in MatMult (RISC-V), with CPI reductions reaching up to 22.3%. These results confirm that front-end stalls represent a significant and addressable bottleneck in in-order architectures.

RQ2 — Which additional performance parameters can be improved by the proposed solution? Beyond IPC improvements, the IFA positively affected several additional microarchitectural metrics. Pipeline stall counts were substantially reduced across all evaluated workloads, reaching reductions of up to 99.34% in CRC32 (RISC-V) and 61.05% in WikiSort (RISC-V). These results demonstrate the effectiveness of the DAU in bypassing stalled instructions while maintaining continuous pipeline utilization.

Instruction-cache miss rates remained stable or exhibited only minor variations, while branch prediction accuracy was preserved in nearly all evaluated scenarios. Importantly, no significant negative performance regressions were observed in IPC or CPI metrics, although branch prediction accuracy showed minor variation in the BasicMath workload, likely attributable to pipeline timing changes introduced by the IFA buffer.

RQ3 — Does the proposed solution incur lower energy overhead compared to strategies? Using the Race-to-Halt energy model [16] and assuming a conservative 3.5% active power overhead associated with the FEB, DAU, and DU structures, the IFA still achieved an estimated net energy reduction of approximately 3.5%.

These results indicate that the execution-time reduction provided by the proposed architecture compensates for the additional hardware cost, validating the IFA as an energy-

efficient enhancement suitable for embedded and low-power processors.

The current evaluation presents two primary limitations: i) the proposed architecture was validated exclusively through cycle-accurate simulation using Gem5. No RTL synthesis or physical implementation was performed, leaving area utilization, timing closure, and post-synthesis power measurements as open research questions; ii) although the selected benchmark suites are representative of embedded workloads, they do not cover real-time operating-system interference, multicore interactions, or multithreaded execution scenarios.

Future work will focus on three main directions: i) the first involves RTL implementation in SystemVerilog and FPGA-based validation using open-source RISC-V cores such as CVA6 or PicoRV32, enabling precise area, timing, and power measurements; ii) the second involves technology-scaling analysis using open-source process design kits to evaluate the overhead of the IFA in advanced semiconductor nodes below 14 nm; iii) future studies will investigate the behavior of the proposed architecture under embedded TinyML workloads, where instruction-level parallelism patterns may further amplify the performance gains observed in compute-intensive benchmarks such as MatMult.

References

[1] C. Kaynak, B. Grot, and B. Falsafi, "Confluence: Unified instruction supply for scale-out servers," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*, 2015.

[2] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*, 2008.

[3] M. B. Breughe, S. Eyerman, and L. Eeckhout, "Mechanistic analytical modeling of superscalar in-order processor performance," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, pp. 50:1–50:26, 2014.

[4] C. Arul Rathi, G. Rajakumar, T. Ananth Kumar, and T. S. Arun Samuel, "Design and development of an efficient branch predictor for an in-order risc-v processor," *Journal of Nano- and Electronic Physics*, vol. 12, no. 5, pp. 05 021–1–05 021–4, 2020.

[5] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA)*, ser. ISCA '99. IEEE Computer Society, 1999, pp. 16–27.

[6] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[7] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, 1995.

[8] G. A. Chacon, "Processor memory system design for performance and security," Dissertation, Texas A&M University, College Station, TX, 2023.

[9] O. J. Santana, A. Falcón, A. Ramirez, and M. Valero, "Dia: A complexity-effective decoding architecture," *IEEE Transactions on Computers*, vol. 58, no. 4, pp. 449–462, 2009.

[10] G. Reinman, B. Calder, and T. Austin, "Optimizations enabled by a decoupled front-end architecture," *IEEE Transactions on Computers*, vol. 50, no. 4, pp. 338–355, 2001.

[11] P. S. Oberoi, "Out-of-order front-ends," Preliminary Proposal, University of Wisconsin-Madison, Madison, WI, 2001.

[12] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An effective alternative to large instruction windows," *IEEE Micro*, vol. 23, no. 6, pp. 20–25, 2003.

[13] R. Shioya, M. Goshima, and H. Ando, "A front-end execution architecture for high energy efficiency," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 419–431.

[14] N. Binkert *et al.*, "The gem5 simulator," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011, pp. 1–7.

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization (WWC-4)*. IEEE, 2001, pp. 3–14.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012.

[17] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[18] S. Mashimo, A. Fujita, R. Matsuo, S. Akaki, A. Fukuda, T. Koizumi, J. Kadomoto, H. Irie, M. Goshima, K. Inoue, and R. Shioya, "An open source fpga-optimized out-of-order risc-v soft processor," in *International Conference on Field-Programmable Technology (ICFPT)*, 2019.

[19] M. McKeown, J. Balkind, and D. Wentzlaff, "Execution drafting: Energy efficiency through computation deduplication," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 432–444.

[20] R. Huerta *et al.*, "Socgpu: Simple out-of-order core for gpgpus," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[21] A. Miyoshi, C. Lefurgy, E. Hensbergen, R. Rajamony, and R. Rajwar, "Critical power slope: understanding the runtime-leakage tradeoff in deep submicron technologies," in *Proceedings of the 16th international conference on Supercomputing*, 2002, pp. 35–44.

[22] A. Bardizbanyan, M. Själander, and P. Larsson-Edefors, "Reconfigurable instruction decoding for a wide-control-word processor," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, 2011.

[23] A. Djupdal, M. Själander, M. Jahre, S. Aunet, and T. Ytterdal, "Optimizing energy efficiency in subthreshold risc-v cores," in *ResearchGate Preprint*, 2020.

Study of PicoTDC Performance for FIT Detector Upgrade in ALICE Experiment at CERN

Łukasz Drzensla, Sebastian Koryciak, Olksandr Savchenko and Paweł Russek
The Faculty of Computer Science, Electronics and Telecommunications
AGH University of Kraków

Kraków, Poland

Abstract—**In this paper, we present the results of a study on the performance of the PicoTDC. PicoTDC is a new Time-to-Digital Converter (TDC) chip developed at CERN for time-of-flight measurements in high-energy physics experiments. Our research aimed to evaluate the feasibility of using the PicoTDC in the upgraded front-end electronics (FEE) of the Fast Interaction Trigger (FIT) detector in the ALICE experiment. The FIT detector provides the trigger signals for the other ALICE sub-detectors and operates under very tight latency constraints. Since the necessary PicoTDC performance data were not available in the published documentation, we decided to derive the required parameters from simulations. The unique contribution of this paper is a detailed explanation of the output signals' timings of PicoTDC. Additionally, we propose a sensor channel allocation scheme that reduces the guaranteed PicoTDC processing time. We derive and present useful analytical formulas for calculating the output latency across different usage scenarios. Finally, we present a case study that assesses the feasibility and the conditions required to meet the latency and data-rate requirements when incorporating the PicoTDC into the upgraded FIT front-end electronics.**

Keywords: PicoTDC, FIT detector, TDC Latency Optimisation

I. MOTIVATION

The A Large Ion Collider Experiment (ALICE) at the European Organisation for Nuclear Research (CERN) in Geneva is dedicated to the study of heavy-ion physics. It is one of the major experiments conducted at the Large Hadron Collider (LHC). The main objective of ALICE is to investigate the conditions that existed in the early universe. During the LHC Long Shutdown 2 (2019–2022), the ALICE multi-detector underwent a major upgrade, during which several significant improvements were implemented, including the installation of a new detector called the Fast Interaction Trigger (FIT). Now, the FIT detector plays an important role in CERN experiments by providing trigger signals for ALICE, as well as beam monitoring and luminosity measurements for the LHC.

For the LHC Long Shutdown 3 (LS3), an upgrade of the FIT front-end electronics (FEE) is planned. The goal of this upgrade is to extend both the precision and dynamic range of time and charge measurements. In addition, as the Texas Instruments time-to-digital converter (TDC) chips used in the current FIT front-end electronics (FEE) are no longer in production; the search for a suitable replacement has started. Meanwhile, the PicoTDC chip, newly designed at CERN, has matured into a verified device and is now available for use in future designs. Therefore, verifying the feasibility of incorporating the PicoTDC into the FIT FEE became an objective, and it is a topic of this paper. The FIT data processing latency is a crucial parameter for the FIT electronics. Unfortunately, the available PicoTDC documentation [1] does not provide sufficient information regarding PicoTDC timing characteristics, such as latency and data throughput.

As a result, a framework for simulating various PicoTDC use-case scenarios within the FIT detector was developed. An RTL model written in SystemVerilog, representing the PicoTDC chip was provided to the authors by the PicoTDC design team. The simulation testbench was configured, tuned, and extended with additional features to represent accurately FIT detector operating conditions. Through modelling, key PicoTDC timing parameters were identified, enabling the definition of appropriate configurations and operating conditions for its use in the FIT readout electronics.

This paper is organised as follows: in Section II we describe PicoTDC features. In Section III, we describe the means to achieve the desired functionality and parameters. In Section IV, the used simulation framework is explained. In section V, the findings about latency are examined. Finally, Section VI, describes how these findings correspond with the FIT detector's electronics.

This work is co-financed in part supported by the Ministry of Science and Higher Education (Agreement Nr 2023/WK/07)

Manuscript received May 22, 2026; revised July 15, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Short
DOI: doi.org/10.64552/wipiec.v12i2.140

II. PICOTDC INTRODUCTION

PicoTDC is a 64-channel TDC ASIC specifically designed by CERN for time-of-flight measurements in high-energy physics experiments [2]. Following the convention of the chip's manual, the term *hit* is used throughout this paper to denote an input event.

A. TDC architecture

PicoTDC is a sampling TDC based on a delay line and a resistive-capacitive interpolator. [1]. The chip operates with a 40 MHz clock, matching the standard clock frequency used across LHC experiments [3]. It features 64 input channels that are divided into four parallel groups of 16 channels each (see Fig. 1). All channels share the same counters and interpolators [1].

B. Modes of operation

PicoTDC supports two modes of operation: *single edge* and *Time Over Threshold* (TOT). The former mode outputs the absolute time of a leading edge, while the latter additionally measures the time between the leading and trailing edge. The time range is strongly tied to the frame format (see Fig. 2). The time range of the single edge mode is 204.68 µs, for TOT, there are two sub- modes, which support pulse widths up to 6.24335 ns and 777.75 ps, respectively. For the considered use case TOT mode was rejected as unsuitable.

C. Reset and time reference

An important property of the PicoTDC is the reset synchronous with the internal 320 MHz clock. Absolute time, measured by PicoTDC, is referenced to an indeterministic point in time after the reset when the counters start counting [4]. So, PicoTDC cannot be synchronised to the referenced zero time mark, and relative time data is only available.

D. Trigger Matching

Many LHC detectors take readouts only in specific time windows. Additionally, they output the time data only if the provided trigger signal is valid for a given window. For this purpose, PicoTDC has a *trigger matching* function. In PicoTDC, each measurement window corresponds to a single output data frame, and all hits outside the window are discarded during post-processing (see Fig. 3). The trigger windows may also overlap, enabling continuous readout operation. However, PicoTDC has a specific requirement that the time from the trigger signal to the beginning of the window is longer than the window length. Both quantities are multiples of 25 ns (PicoTDC internal clock is 40 MHz) [1]. The operating modes with and without the trigger matching function are called *triggered* and *triggerless*, respectively.

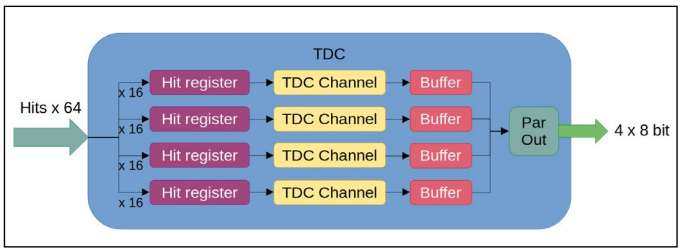


Figure 1. Simplified PicoTDC internal diagram depicting its parallel channel groups.

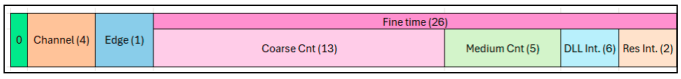


Figure 2. PicoTDC frame in single edge, triggerless mode.

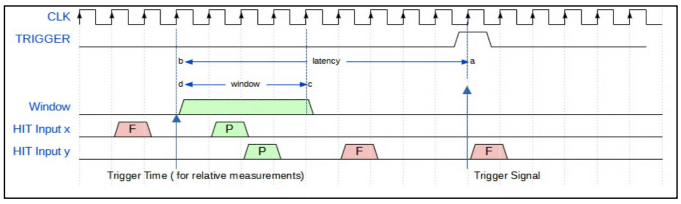


Figure 3. Trigger matching waveforms. Taken from [1].

III. CHOICE OF APPROACH

Many ALICE sub-detectors don't have the latency constraint. These detectors can aggregate hits from many bunch crossings. This allows for better utilisation of bandwidth for communication with FLP as hit data is collected with headers and trailers. PicoTDC is well-suited for this demand. PicoTDC can take data during time windows which can be much longer than a single BC period. The length of the window is configurable.

Windows of interest (later referred to as *trigger windows*) are validated by the trigger signal (see Fig. 4), which can come with the delay of the configured trigger latency.

The fundamental constraint restricted by the PicoTDC chip design is that the trigger latency shall be greater than the window width. Both the width and latency are expressed as multiples of 25 ns (period of 40 MHz LHC clock).

To optimise latency, we want to set the window duration to 25 ns and trigger latency to 50 ns. However, these parameters are not feasible because they lead to overflow of output data buffers. The output data frame consists of headers (HDR, 4 bytes), data (TDC, *number_of_hits* × *4 Bytes*) and trailer (TRL, 4 Bytes). Minimum size of frame is 12 Bytes. With the said parameters, this leads to *12 Bytes* × *40 MHz* = *480 MB/s*. Meanwhile, the PicoTDC output data rate is 320 MB/s. So the theoretical minimum latency setup configuration leads to buffer overflow. A feasible configuration is 50 ns for the window and 75 ns for the trigger latency. For this configuration, we have *12 Bytes* × *20 MHz* = *240 MB/s*. However, the latency is 155 ns, a demonstrated in Fig. 4.

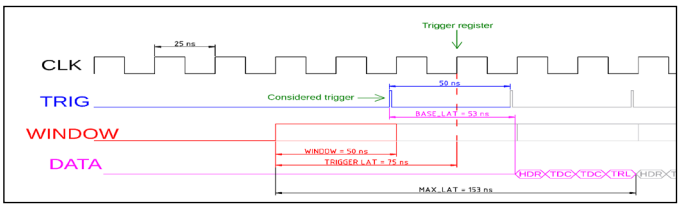


Figure 4. Trigger window, trigger latency and total latency in triggered mode. The trigger window is set to 50 ns, and the trigger latency is set to 75 ns. This is the lowest-latency valid configuration. Here, it is assumed that two hits arrived during the considered window (worst-case scenario), resulting in two TDC measurements on the data line.

This latency can be shortened if we abandon the PicoTDC triggered mode and use our custom configuration. In our proposed configuration LHC is fed into one of the PicoTDC input channels, and the relative hit time is calculated from the PicoTDC output data, but subtracting the clock time and hit time (see Fig. 5).

To verify the guaranteed latency of the designed system, a worst-case approach was chosen. This means that in the given BC, every PicoTDC channel is hit.

Latency of PicoTDC can be defined as the time elapsing from the leading edge of the first hit in BC to the trailing edge of the last bit of the last output data frame related to this BC.

IV. METHODOLOGY

The proposed idea was simulated using the PicoTDC RTL model that mimics the behaviour of the circuit in tools such as Vivado. It was kindly provided to the authors by CERN PicoTDC team member. The simulations revolved around latency.

To simply and accurately manipulate hits in the simulation model, the hit generator, from the testbench provided in the model, was replaced with a custom one that allows dividing hits among the input channels of a TDC in the later described *uniform channel allocation* manner. The new hit generator allows single-shot hits simulation for the worst-case scenario when multiple hits arrive at the inputs during the same evaluation window. The new hit generator is written in SystemVerilog and was extensively used throughout the research conducted for this work.

V. RESULTS OF LATENCY SIMULATION

A. *Evaluation window*

Our simulations show that PicoTDC collects hits before they are sent to the data output interface. The term *evaluation window* is introduced to describe this time period during which PicoTDC collects hits. The duration of this window is 12.5 ns. Also, data frames are emitted at discrete intervals corresponding to multiples of this duration - input hits are effectively grouped within 12.5 ns intervals, and the corresponding data frames are transmitted at a fixed offset relative to the start of each interval (see Fig. 6, 8).

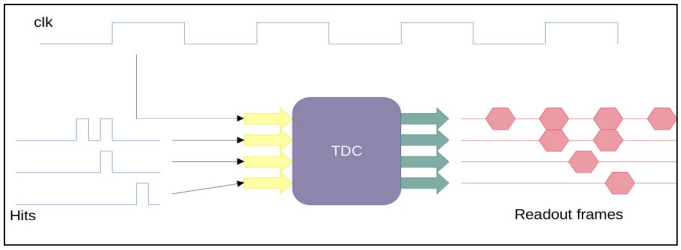


Figure 5. Proposed solution to achieve relative time measurements without using the trigger matching function.

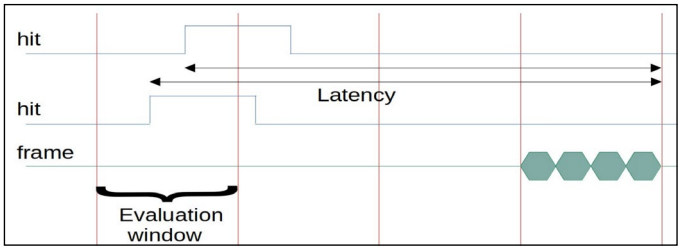


Figure 6. The figure depicts two scenarios for a single device hit. Note that the latency varies based on where the hit falls within the evaluation window.

Consequently, the latency of PicoTDC is not static - it varies because hits can happen at any time within the evaluation window. When a hit occurs just before the edge that triggers evaluation, the latency is smaller. When a hit happens further away from the edge, the latency increases. The waveform of latency over time from circuit startup is thus saw-shaped (see Figure 7).

B. *Output latency*

Total latency consists of base latency and data transmission period.

The *base output latency* (L_{base}) term refers to the interval from a hit to the beginning of the first output frame. Simulation results indicate that this latency varies between 52.5 ns and 65 ns. This is due to the relation of an incoming hit to the evaluation window.

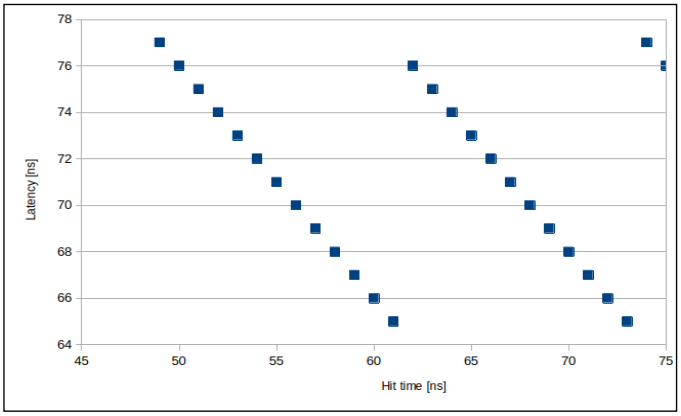


Figure 7. Latency variation forms a saw-like waveform. This pattern continues as subsequent hits arrive at the circuit inputs. The horizontal axis represents the time at which the hit arrived, from the start of the simulation.

C. *Uniform channel allocation*

As already mentioned, PicoTDC has 64 channels that are divided into four parallel channel groups of 16. Hit channels can be distributed across PicoTDC input channels in a strategic manner to minimise latency through parallelism, later referred to as uniform channel allocation (see Fig. 9, 10, 11). The number of input channels assigned to each channel group shall be equal. If the number of input channels is not divisible by the number of channel groups, the difference between maximum and minimum number of input channels assigned to each group shall not exceed 1. For N channels connected to one channel group, the maximum latency of a group can be expressed by the following equation:

$$L_{group\_max} = L_{base\_max} + N \times T_{frame}, \quad (1)$$

where L_{base_max} represents the maximum value of base output latency and T_{frame} represents the frame length.

The maximum latency of the chip in such a use case is then equal to the largest latency among the four groups.

D. *Maximum hit rate*

The maximum accepted hit rate of continuous periodic events is not given in the PicoTDC reference manual, so it was determined through analysis and simulation. While the manual states the minimum time interval between hits for them to be distinguished by the TDC at 781 ps [1], readout of hits continuously incoming with this frequency is not possible. The main limitation is the output interface throughput. Each channel group is capable of transmitting 12.5 ns long frames serially, resulting in a frame rate of 80×10^6 frames/s. Therefore, the maximum hit rate for a channel group for hits incoming constantly and periodically is 80×10^6 hits/s per channel group. A higher rate is acceptable over a short period of time until the buffers get filled.

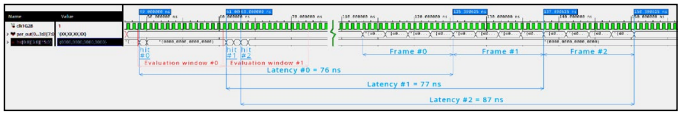


Figure 8. Example waveforms of hits in a single channel group. *clk1G28* is a simulation-only 1.28 GHz clock that defines the beginning and end of an evaluation window. *par_out* is the output interface of the chip, and hit is the 64-channel input interface. Hits 1 and 2 occur in the same evaluation window (window 1), so frame 2 must wait for frame 1, effectively increasing the latency of frame 2. Values are rounded for clarity.

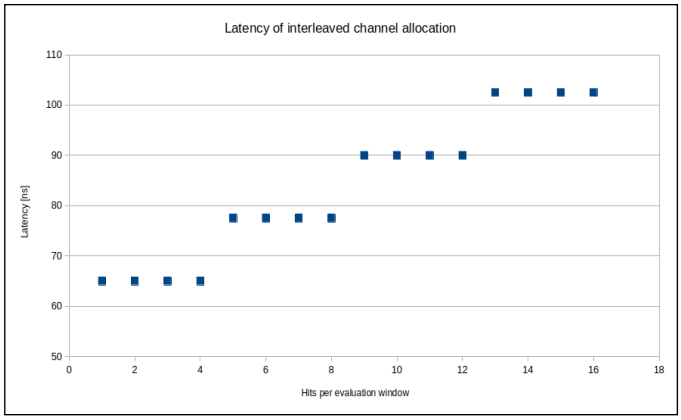


Figure 9. With uniform allocation, described in V-C, the PicoTDC processes hits in parallel across its four channel groups. Consequently, latency does not scale continuously; instead, it jumps by an additional frame length only when the total number of simultaneous hits across the chip exceeds a multiple of four (e.g., 5, 9, or 13 hits), forcing multiple channels to share the same group output. The hits' relation to the evaluation window was the same for all cases.

E. *Time reference problem*

For utilisation within FIT readout electronics, a refer- ence to the LHC clock has to be provided. The solution proposed in this paper is to connect the LHC clock to one input of the TDC. When a hit comes, a simple subtraction allows to determine the hit time relative to the clock. Then, in postprocessing, the hit can be normalised to determine its relation to the LHC clock using the following equation:

$$t_{normalised} = ((t - t_{lhc\_clk} + 0.5T) \bmod T) — 0.5T, \quad (2)$$

where t is the measured hit time, t_{lhc_clk} is the timestamp of the LHC clock measured by the TDC and T is the period of the clock. The equation is also valid in the event of counters rollover, as the 25 ns LHC clock period is below the PicoTDC time range in single edge mode (see Section II-B) and therefore it was chosen as the operating mode for the FIT FEE use case.

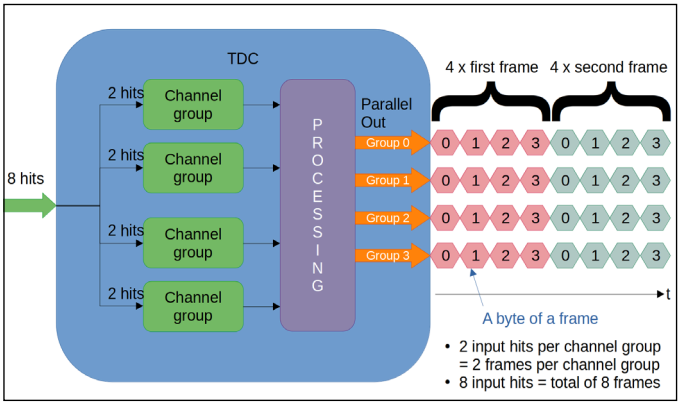


Figure 10. Utilisation of channel group parallelism with uniform channel allocation.

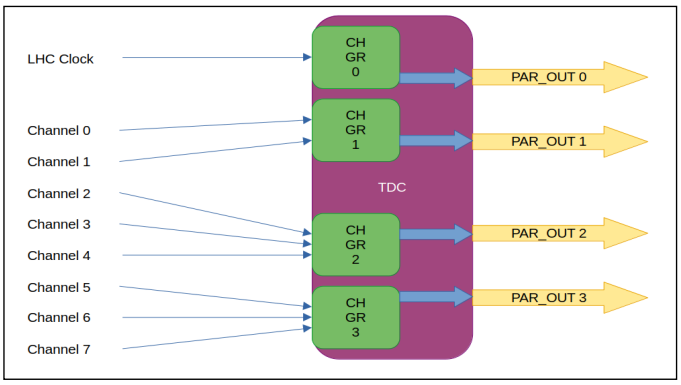


Figure 11. Uniform channel allocation for FV0. Note that channel group 0 is excluded from the input channels allocation.

Clock synchronisation timestamps must be continuously monitored. Because the signal is periodic, the frames produced by the signal will also be periodic. Since the frequency is 40 MHz, and the maximum allowed is 80 MHz; there is only space for one frame between each clock cycle. However, it cannot be guaranteed that spontaneous hits will not come in many at once, producing many frames. Many frames will result in obstruction of the clock timestamps. Then the synchronisation would fail. Therefore, it shall be assured that no hits will come to the same channel group as clock timestamps, so none of the channels in the same channel group as the clock input can be utilised. Thus, 16 channels are lost to perform this operation. This significantly reduces the latency capabilities of this solution, as only three, not four, measurements can be performed in parallel.

VI. FIT DETECTOR USE CASE

A. *FIT FEE*

FIT detector consists of three sub-detectors: FV0, FT0 and FDD. For each of them, the FEE consist of two major units: a Trigger and Clock Module (TCM) and a number of Processing Modules (PM) (see Fig. 12).

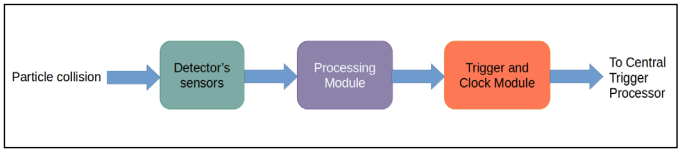


Figure 12. Simplified diagram of FIT front-end electronics

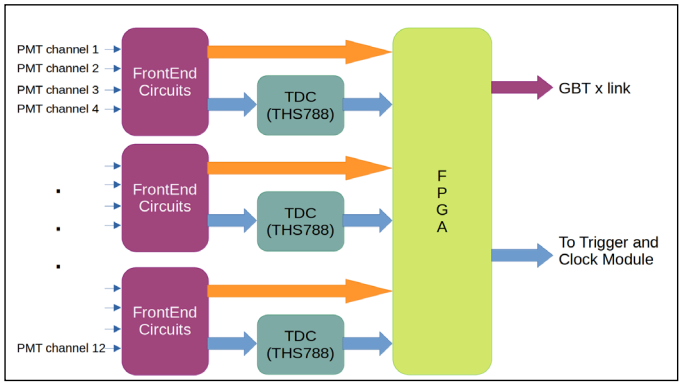


Figure 13. Simplified diagram of current, pre-upgrade, PM design. Based on: [7].

The component of FIT's electronics responsible for providing the time of the event is called a Processing Module (PM). This element features 12 channels and uses three Time to Digital Converters (TDC) to provide the time a particle hit detector's photomultipliers (PMT) to other readout systems (see Fig. 13). This work concentrated on a case study for FV0, the largest of three subdetectors of FIT, which utilises only 8 channels per PM. One of the objectives of the research was to check whether it is possible to use only a single TDC chip while maintaining the required latency.

The latency for the entire data path of FIT's readout electronics must not exceed 425 ns [5]. In this time budget, a 181 ns delay is introduced by the PMs and the rest by coaxial cables and other components [6]. A requirement was established that any replacement Time to Digital Converter must achieve a latency equal to or lower than that of the currently used THS788, measured at 116.0 ns.

B. *FV0 case study constraints*

The research revolved around an example concerning the FV0 detector. FV0 is divided into 48 sections, with each section corresponding to one data channel. These channels are then connected to 6 PMs, distributed uniformly - 8 channels are connected to each PM [7]. In the current design, there are three TDCs per PM, and four channels are connected to one TDC. Simulations were conducted to verify whether one TDC is sufficient to meet the latency requirements with all 8 input channels utilised.

C. *Proposed solution for FV0*

An example solution of uniform channel allocation, with LHC clock reference, for the FV0 use case, is depicted in the Fig. 11. FV0's 48 channels are distributed evenly among 6 PMs. The following considerations assumed a single PicoTDC chip

per PM. The simulated maximum latency for this system was 102 ns and can be expressed with an equation:

$$L_{total} = L_{base\_max} + 3 \times T_{frame}, \quad (3)$$

as there might be as many as three hits incoming at a channel group during the same evaluation window. The result of 102 ns fits within the 116 ns limit.

However, another problem arises. The maximum hit rate per channel of the readout electronics is determined by the frequency of the 40 MHz LHC clock. During each cycle, there is a 5 ns wide time window within which hits will be recorded. At maximum, there are three input channels connected to a single channel group. Assuming that every LHC cycle (40 MHz clock), a hit may come, the maximum per channel group hit rate is

$$3 \text{ hits} \times 40 \text{ MHz} = 120 \times 10^6 \text{ hits/s} \quad (4)$$

exceeding the limit of 80×10^6 hits/s. While this worst-case scenario is quite unlikely, a system like FIT cannot allow any sort of unreliability. Therefore, a single PicoTDC chip is not suitable for the use case. If two chips were used, the requirements would be fulfilled successfully. After utilising the first channel group of both TDC's for the LHC clock a total of 6 groups are left, so there will be 2 channels allocated in each group, resulting in 89.5 ns maximum latency. The findings are summed up in Table I.

TABLE I. SUMMARY OF DESIGN CONSTRAINTS AND PICOTDC-BASED SYSTEM SUITABILITY (FV0)

Configuration	*Resolution*	*Latency*	*Req. hit rate / ch group*
1× PicoTDC	3.05 ps	102 ns	120×10^6 hits/s
2× PicoTDC	3.05 ps	89.5 ns	80×10^6 hits/s

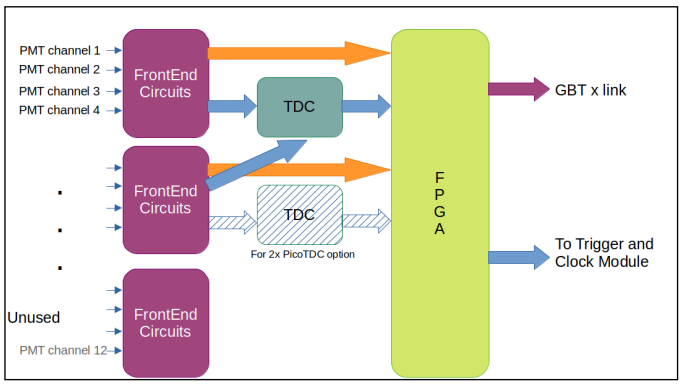


Figure 14. Simplified diagram of a PM with one or two PicoTDC chips. (FV0 use case)

VII. SUMMARY

The analysis of the PicoTDC circuit shown in this paper proved the chip to be suitable for FIT electronics upgrade in the terms of latency. However, to guarantee reliable functioning in the worst case scenario, in terms of hit rate, two chips are required for the FV0 case, but at the same time, they are capable of covering all the 12 PM channels (for other detectors than FV0). This allows scarcer input channel distribution and therefore reduces the maximum effective hit rate. However, using multiple chips significantly increases the cost. However, the integration is still possible; therefore, PicoTDC is viewed as a serious candidate for THS788 replacement.

REFERENCES

[1] CERN, picoTDC: Picosecond time to digital converter manual, CERN, Jan. 2024, for picoTDC Version 2.0.

[2] J. Christiansen, M. Horstmann, L. Perktold, and J. Prinzie, "pi- coTDC: Picosecond TDC for HEP," in IEEE International Symposium on Circuits and Systems (ISCAS), Monterey, CA, USA, May 2024

[3] S. Altruda et al., "PicoTDC: A flexible 64 channel TDC with picosecond resolution," in JINST 18 P07012, 2023.

[4] CERN PicoTDC Design Team, "Private communication regarding PicoTDC architecture," Personal communication, Dec. 2024.

[5] M. Słupecki, "The fast interaction trigger for the ALICE upgrade," PhD Thesis, 2020.

[6] D. Serebryakov, "FIT electronics," Dec. 2020, presentation, CERN.

[7] ——, "FIT electronics," Apr. 2019, presentation, CERN

Design and Implementation of Three-stage RISC-V Microprocessor for Education

Jan Ber
The Faculty of Computer Science, Electronics and Telecommunications
AGH University of Krakow
Krakow, Poland

***Abstract*—In many computer science and embedded systems courses, processor architecture is typically introduced using either the classical five-stage RISC pipeline or the two-stage AVR pipeline. While these architectures provide a strong introduction to bare-metal programming and computer organization, they do not necessarily represent the most practical or accessible approach for teaching modern processor architectures. In particular, traditional five-stage pipelines often introduce considerable control and forwarding complexity, while compact microcontroller architectures may obscure fundamental pipeline behavior and instruction flow.**

This paper presents X3S, a work-in-progress 32-bit pipelined RISC-V softprocessor core originally developed as author's engineering thesis. The design adopts a simple three-stage pipeline intended to balance architectural clarity, implementation complexity, and achievable operating frequency on Field Programmable Gate Array (FPGA) devices. X3S implements the RV32I base integer instruction set together with the M extension for multiplication and division operations. Multi-cycle arithmetic accelerators are integrated into the execution pipeline using a handshake-based interface.

***Index Terms*—RISC-V, RV32I, soft processor, microprocessor, FPGA, pipelined processor**

I. Introduction

Effective software development, particularly for students in embedded systems education and computer science, requires more than familiarity with high-level programming abstractions. In practical systems, performance, determinism, and resource usage are directly determined by the underlying hardware architecture. Consequently, understanding how software maps onto hardware execution is critical for writing efficient and reliable bare-metal code. This is especially relevant in embedded systems education, where students must reason about execution on in-order processors, constrained memory and compute resources, and timing-sensitive peripherals.

However, educational approaches to processor architecture do not always reflect this requirement in a balanced way. In many curricula, instruction is based either on compact microcontroller architectures such as AVR or on classical five-stage RISC pipelines. These two approaches represent opposite ends of the design space in terms of complexity and visibility of internal execution.

AVR-based systems offer a very accessible programming model with minimal architectural overhead, which makes them suitable for introducing low-level programming concepts [1]. However, their execution model and memory organization, including the separation of program memory, data memory, and peripheral address spaces (e.g., memory-mapped I/O such as GPIO), provide limited insight into timing behavior and execution dynamics present in more recent embedded processors [2]. In particular, instruction execution is largely sequential with overlap limited to fetch and execution, which makes for highly predictable timing and abstracts away effects such as variable memory latency and wait states. Consequently, the relationship between instruction execution, memory access, and peripheral interaction is partially obscured, as illustrated in Fig.1.

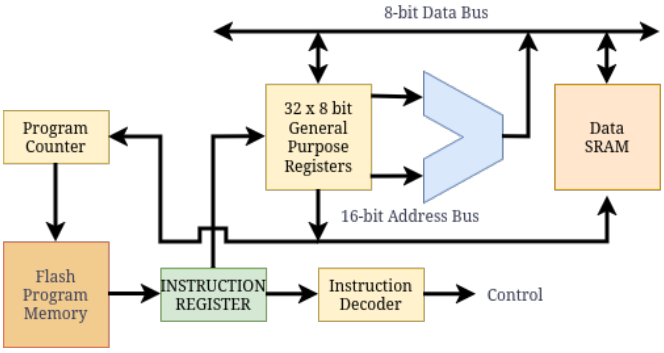


Fig. 1. Simplified dataflow in AVR processors.

In contrast, classical five-stage RISC pipelines are often used as a teaching model to explain instruction-level parallelism and pipelined execution. While this abstraction is useful for teaching the principles of pipelining, it does not directly correspond to many contemporary embedded processor architectures. Modern microcontroller cores, such as those in the ARM Cortex-M family, typically employ pipeline organizations that differ from the classical five-stage model. For example, Cortex-M3 and Cortex-M4 cores use a simple three-stage pipeline (fetch, decode, execute), while larger designs such as Cortex-M7 introduce deeper pipelines with additional buffering, speculation, and dual instruction issue [3].

Author's intention in this paper is to argue that a three-stage pipeline provides a more effective educational balance between architectural clarity and implementation complexity than either compact microcontroller models or classical five-stage RISC pipelines. To this end, X3S processor is used as a case study to demonstrate this design point.

Manuscript received May 24, 2026; revised August 10, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Short
DOI: doi.org/10.64552/wipiec.v12i2.150

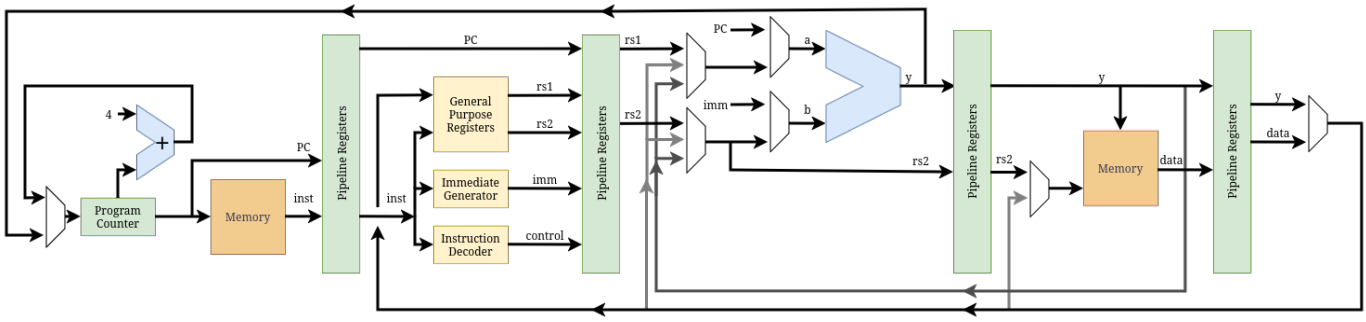


Fig. 2. Dataflow of a typical five-stage RISC-V processor.

TABLE I
EXECUTION TIMELINE IN A 3-STAGE PIPELINE (E.G., CORTEX-M3/M4-CLASS CORES).

Cycle	IF	ID	EX
N	inst0		
N+1	inst1	inst0	
N+2	inst2	inst1	inst0
N+3	inst3	inst2	inst1
N+4	inst4	inst3	inst2

II. FIVE-STAGE PIPELINE

The classical five-stage pipeline is a well-established abstraction used to introduce instruction-level parallelism and pipelined execution. While it has historically been a common design model, it is now primarily used as a teaching reference rather than an accurate representation of most contemporary embedded processor architectures. In this work, it serves as a baseline model for explaining pipeline concepts and for comparison with simpler pipeline organizations.

This section uses a standard RISC-V five-stage datapath, as commonly presented in literature [4], consisting of instruction fetch (IF), instruction decode (with register read) (ID), execute (EX), memory access (MEM), and write-back (WB), as illustrated in figure 2. The model represents an in-order pipeline where multiple instructions are processed concurrently at different stages as shown on Table II.

TABLE II
EXECUTION TIMELINE IN A FIVE-STAGE RISC-V PIPELINE.

Cycle	IF	ID	EX	MEM	WB
N	inst0				
N+1	inst1	inst0			
N+2	inst2	inst1	inst0		
N+3	inst3	inst2	inst1	inst0	
N+4	inst4	inst3	inst2	inst1	inst0
N+5	inst5	inst4	inst3	inst2	inst 1

A. Data hazards and forwarding

While instruction pipelining improves throughput, it introduces several classes of hazards that can prevent the next instruction from executing in the intended cycle. These hazards arise from resource conflicts (such as register file ports or execution units), control flow changes, and most importantly in the context of this paper, data dependencies between instructions.

A *data hazard* occurs when an instruction depends on the result of a previous instruction that has not yet completed its passage through the pipeline. In a classic five-stage pipeline, this is particularly relevant when a register is read during the instruction decode (ID) stage before a preceding instruction has written its result back in the write-back (WB) stage. As a result, the required operand may still be in flight, and the value read from register file is invalid.

A common example is a read-after-write (RAW) dependency:

```
inst1: add x1, x2, x3
inst2: sub x4, x1, x5
```

Here, `inst2` requires the value of `x1` produced by `inst1`. In a non-pipelined design, this dependency is trivially resolved because `inst1` completes before `inst2` begins. However, in a pipelined processor, `inst2` may reach the ID stage before `inst1` has written its result back to the register file. This creates a hazard where the decode stage would read a stale value of `x1` unless a forwarding mechanism is used.

In a five-stage pipeline, implementing data forwarding requires additional multiplexing logic to route results between pipeline stages. Three forwarding multiplexers are typically required: two in the EX stage and one in the MEM stage.

In RISC-V processors, forwarding logic must be integrated with the operand selection already present in the ALU datapath. ALU inputs are selected between register operands (`rs1`, `rs2`), immediate values, and the program counter (PC). Supporting forwarding therefore either increases the width of the operand selection multiplexers or introduces additional multiplexing stages in series. Although the required hardware is modest, wider multiplexers and cascaded selection logic increase combinational delay and extend the processor critical path.

B. Do we need WB stage?

The WB stage is primarily responsible for selecting between the ALU result and memory read data before writing the selected value back to the register file. Functionally, this stage often consists only of a small multiplexer together with the register file write interface. On FPGAs designs, memory

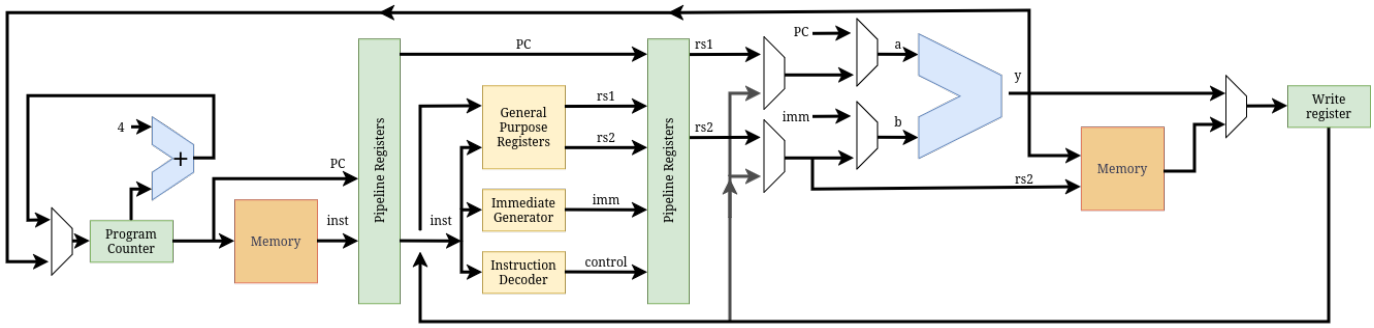


Fig. 3. Proposed pipeline organization with combined EX-MEM-WB stages.

access is fast compared to wide multiplexer networks, making it practical to merge the write-back selection into the MEM stage. This reduces the number of forwarding sources, making forwarding multiplexers in EX stage shallower since data only needs to be forwarded from a combined MEM/WB stage.

C. Do we need a MEM stage?

A similar argument can also be made for the MEM stage, memory access occupies a dedicated stage positioned between execute and write-back. However, load instructions already require the pipeline to stall whenever memory data is not immediately available, meaning that the MEM stage does not necessarily provide fully independent pipeline progress.

For load instructions, the EX stage can compute the effective address and issue the memory request by storing the address in a register connected to the memory interface. The pipeline then stalls until the requested data is returned. Once available, the loaded value can be written directly into a result register used both for forwarding and for register file write-back.

Store instructions can be handled similarly by storing the computed address together with the value to be written into dedicated registers associated with the memory interface. Since store operations do not immediately produce values required by subsequent instructions, execution may continue while the memory subsystem completes the write transaction.

Under this organization, the EX stage effectively becomes the final active stage of the pipeline. Arithmetic results and completed memory responses are written into a shared result register that feeds both the forwarding network and the register file. Proposed organization is shown on figure 3.

III. X3S softprocessor

X3S is a 32-bit RISC-V soft processor core implementing the RV32I base integer instruction set together with the M extension for integer multiplication and division. The design is released as an open-source project and was originally developed as engineering thesis of the author [5].

X3S processor implements an in-order pipeline organized into three stages: instruction fetch (IF), instruction decode (ID), and execute (EX). The design follows a strict single-issue model in which the IF stage is capable of fetching one instruction per clock cycle, provided that older stages aren't stalled. The ID stage performs instruction decoding, register file access, and immediate generation, but does not participate in execution or memory interaction beyond operand preparation. As a result, the ID stage is purely combinational in nature from a datapath perspective, with state changes occurring only through pipeline registers between stages.

The EX stage represents the architectural center of the processor and is responsible for all operations that modify architectural state. This includes arithmetic and logical computation, branch condition evaluation, address generation, and memory transactions. Importantly, only the EX stage is permitted to update the program counter, meaning that control-flow redirection (jumps and taken branches) is resolved exclusively at this stage. This introduces a fundamental control hazard: when a branch or jump is taken, the instructions already fetched in IF and decoded in ID must be invalidated, resulting in at least three cycle control penalty.

Memory operations are mediated entirely by the EX stage using a request–response handshake with an external memory system. Store instructions are issued in the same cycle in which the effective address and write data become available. If the memory interface does not assert backpressure, the pipeline may continue to accept consecutive store instructions at a rate of one per cycle. Load instructions, in contrast, always introduce a stall condition and hold pipeline progress until valid data is returned from memory.

A. Instruction Fetch (IF)

The instruction fetch stage maintains the program counter (PC), issues instruction memory requests, and forwards fetched instructions to the decode stage. In normal operation, next PC is incremented by four each cycle, corresponding to the fixed 32-bit instruction width, fetching one instruction per cycle.

On a pipeline stall, the PC and IF output registers are frozen, and no new memory requests are issued. The most recently fetched instruction and its PC are held at the IF–ID boundary, allowing the decode stage to continue operating on a valid instruction until the stall is released. Execution then resumes without re-fetching or re-decoding.

Control-flow redirection is resolved using a branch signal from the execute stage. When a branch or jump is taken, the PC

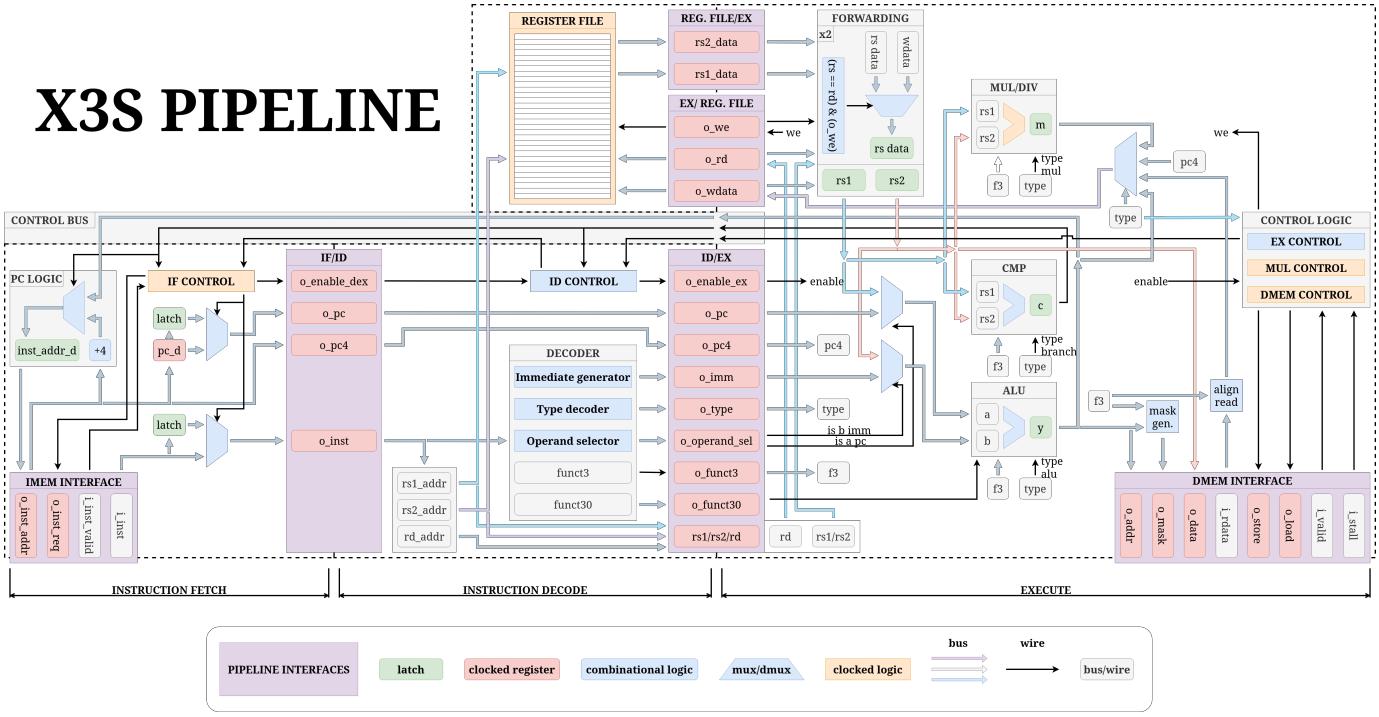


Fig. 4. Dataflow in the X3S processor.

is updated with the target address and any previously fetched instructions are invalidated.

B. Execute (EX)

The Execute stage (EX) acts as the integration point for all non-trivial datapath behavior in the X3S processor. Rather than introducing additional pipeline stages, EX aggregates several functional subsystems and resolves their interactions through internal control logic. Its primary architectural role is therefore not only computation, but also coordination of dependent execution units.

EX contains the operand forwarding network used to resolve read-after-write hazards. This forwarding logic is coupled with the operand selection stage and ensures that dependent instructions can execute without introducing additional pipeline stalls, provided that results are already available within the stage. The execution datapath also includes the Arithmetic Logic Unit(ALU) and comparator (CMP) logic used for arithmetic, logical and address generation, and branch condition evaluation respectively.

EX also acts as the exclusive owner of all external execution-side interface, the data memory interface is fully controlled within this stage. Both load and store transactions are initiated from EX, and all associated control signals (address, write data, and byte enable masks) are generated locally. Load operations are additionally governed by an internal control state machine that tracks request issuance and response validity, allowing the pipeline to be stalled only when required by memory latency.

EX also integrates the multiplication and division unit (M-unit). This block is decoupled from the main combinational datapath and interacts with EX through a handshake protocol based on start and completion signals. From the perspective of EX, the M-unit behaves as a latency-variable execution accelerator. When used, EX stalls the pipeline until completion is signaled, which ensures correct serialization at expense of throughput (since under this model, M-unit cannot be pipelined).

Overall, EX serves as the execution stage in which all architectural effects are resolved, including forwarding, memory transactions, and accelerator-style execution units within a single execution unit.

C. Verification

The X3S processor was verified using both simulation and FPGA implementation. Simulation-based verification was performed using `Icarus Verilog` and `vvp`. The testbench instantiates the processor together with a behavioral memory model and supports automated execution of RISC-V ISA tests [12]. Test completion and status are signaled via memory-mapped writes, enabling batch verification.

Hardware validation was carried out on a *Spartan-7 XC7S50-1* FPGA as part of the X3S-Hydrogen microcontroller, which integrates the core with memory and Universal Asynchronous Receiver Transmitter (UART). X3S-Hydrogen has been synthesized using Vivado Design Suite, and verified on FPGA with clock frequencies up to 100 [MHz]. The maximum clock frequency is limited by the critical path is in the EX stage and is caused by operand selection and forwarding multiplexers

before the ALU. Resource usage by X3S processor is shown on the Table III.

TABLE III
RESOURCE USAGE BY X3S PROCESSOR IMPLEMENTED IN X3S-HYDROGEN.

Module	Slice LUTs	Slice Registers	BRAM	DSP
X3S-Hydrogen	1697	1176	8	3
Pipeline	1574	935	–	3
IF	128	227	–	–
ID	600	127	–	–
EX	721	517	–	3
MUL	245	134	–	3
DIV	325	202	–	–
PeripheralBus	4	55	–	–
UART	74	126	–	–

IV. Conclusion and further work

This paper presented X3S, a 32-bit RISC-V soft processor implementing a simplified three-stage pipeline designed for educational use. The architecture was motivated by the limitations of both classical five-stage pipelines and minimal process models in teaching. By merging and reorganizing pipeline stages, X3S reduces structural complexity while preserving essential concepts such as instruction-level parallelism, data hazards, and control flow effects. The implemented design demonstrates that a three-stage organization can achieve a practical balance between clarity and hardware efficiency on FPGA platforms.

Further improvements to the X3S processor could be made by relocating parts of the operand selection logic from the EX stage to the ID stage, which would leave only the forwarding multiplexers in the execution stage. In addition, introducing an extra adder in the ID stage would allow address calculations for jumps and memory operations to be performed one cycle earlier, simplifying the EX-stage control logic. This change would require operand forwarding within the ID stage; however, this can be supported by implementing the register file using BRAMs, whose access time is typically comparable to or faster than the additional combinational adder, ensuring that register reads and address generation do not become the critical timing bottleneck.

References

[1] Microchip Technology Inc., *Avr instruction set manual*, 8-bit AVR architecture reference, n.d. Accessed: May 21, 2026. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-InstructionSet-Manual-DS40002198.pdf

[2] Microchip Technology Inc., *Atmega328p datasheet*, AVR microcontroller documentation, n.d. Accessed: May 21, 2026. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega328P_Datasheet.pdf

[3] Arm Ltd., *Armv7-m architecture reference manual*, Cortex-M architecture reference, n.d. Accessed: May 21, 2026. [Online]. Available: https://developer.arm.com/documentation/ddi0403/latest/

[4] S. L. Harris and D. M. Harris, *Digital Design and Computer Architecture: RISC-V Edition*, 2nd ed. Morgan Kaufmann, 2021.

[5] Jan Ber. "X3s risc-v soft processor core." Open-source FPGA soft-core implementation, Accessed: May 21, 2026. [Online]. Available: https://codeberg.org/jber/little-risc-v-softcore

A Low-Power ADC-Free Resistive Sensor Readout Circuit for µC-Based Microsystems

João Maia Aguiar
INESC TEC, Faculdade de Engenharia
Universidade do Porto
Rua Dr Roberto Frias, 4200-465 Porto, Portugal
joao.m.aguiar@inesctec.pt

José Machado da Silva
INESC TEC, Faculdade de Engenharia
Universidade do Porto
Rua Dr Roberto Frias, 4200-465 Porto, Portugal
jms@fe.up.pt

Abstract — **Implantable microsystems are required to show low-volume and low-power consumption, while ensuring high reliability and safe long-term operation. This work addresses the development and evaluation of a compact readout circuit for a piezoresistive force sensor intended for integration in an active oral prosthesis. The approach proposed here can be included in the direct-to-microcontroller interface methodologies, but relies on a sigma-delta-like measurement process. Its performance has been experimentally compared to conventional Wheatstone bridge and simple voltage-divider topologies, showing, respectively, 76% and 44% reductions in power consumption, while ensuring 8 times better resolution.**

Keywords- Resistive sensor; ADC-free readout; low-power.

I. INTRODUCTION

The oral cavity plays an essential role in mastication, swallowing, speech, and postural regulation. Teeth, muscles, temporomandibular joints, and the nervous system act together to regulate masticatory bite force, depending on the sensory feedback from oral mechanoreceptors, particularly periodontal mechanoreceptors. These detect force, direction, and duration of tooth loading and transmit this information to the central nervous system [1,2]. This feedback is essential for efficient chewing and to prevent excessive mechanical stress. Tooth loss reduces or eliminates the natural periodontal feedback pathway, leading to reduced perception of biting and chewing forces. Excessive masticatory loading has been associated with fixed-prosthesis complications, including ceramic fracture, screw loosening, peri-implant bone loss, and premature implant failure [2,3].

The application targeted here concerns the development of an active oral prosthesis, featuring pressure-sensing and stimulation circuits meant for sensory feedback restoration. Biofeedback-based approaches have already been explored for managing masticatory muscle activity in awake bruxism [4], but no developments have been proposed regarding the restoration of biting and chewing proprioception. Embedding a sensing and stimulation electronic microsystem within such a prosthesis imposes severe constraints on its characteristics: it must be compact, thermally safe, and show low-volume/low-power to ensure comfort and integrate easily into the prosthesis.

This work has been supported by Portuguese Fundação para a Ciência e Tecnologia under PhD grant 2025.05957.BDANA.

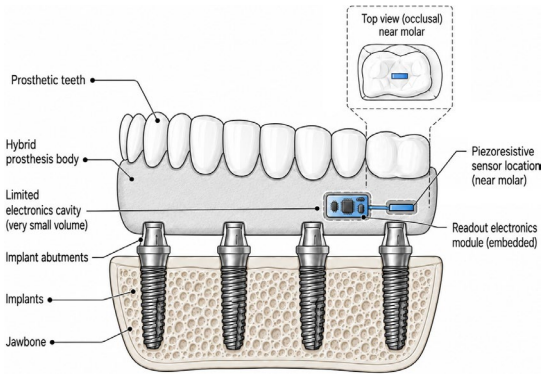


Figure 1. Proposed active prosthesis, showing the placement of the force sensor and readout electronics.

Fig. 1 illustrates the integration scenario being considered in this work. For the pressure-sensing stage, piezoresistive sensors are being used due to their simple structure, small size, and direct resistance variation under mechanical load. The sensor being used is a HAPTICA semiconductor strain gauge integrated on a flexible substrate, with nominal resistance of $5 \pm 0.5\ k\Omega$, dimensions of $\approx 1.524 \times 0.838 \times 0.203\ mm^3$, and a specified gauge factor of $GF = 175 \pm 10$. Its reduced size, flexible support, and high sensitivity make it suitable for integration in the prosthesis structure being designed.

Conventional resistive sensor interfaces often rely on constant voltage Wheatstone Bridge (WB) and simple resistive voltage divider (VD) configurations. However, these topologies imply: the former ones, the need for extra high-accuracy components, high-resolution analog-to-digital converters (ADCs), an instrumentation amplifier; the second ones show lower sensitivity and higher susceptibility to supply voltage noise and loading errors, and allow for the use of single-ended ADCs. Also, lower power consumption can be achieved with VD if a high series resistance is used, but eventually that implies the need for a low-power signal amplification stage prior the ADC to ensure the required dynamic range is captured [5]. Current biased WB and VD sensor interfaces can be adopted, but these imply extra circuitry to implement the current source.

Manuscript received May 25, 2026; revised August 04, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Short
DOI: doi.org/10.64552/wipiec.v12i2.142

TABLE I. CHARACTERISTICS AND FEATURES OF RESISTIVE SENSOR ADC-FREE READOUT CIRCUITS.

Ref.	# Components	Resolution	Sensitivity	Sensor Range
[12]	8	20 counts/Ω	0.05 Ω at 1 MHz	1–19 Ω
[13]	8	20 counts/Ω	0.05 Ω at 1 MHz	0.5 kΩ–4.5 kΩ
[14]	10	32.8 counts/Ω	0.03 Ω at 656.25 kHz	80–150 Ω (Pt100), 1 kΩ–1 MΩ (Pt1000)
[15]	9	—	Current boosting (21.1–49.4×)	3.63 kΩ–488.2 kΩ
[16]	9	—	Resistance-to-time conversion	RTD equivalent range
[17]	4	8-bit	Adaptive DAC-based readout	Conductive polymer gas sensors
[this work]	3	42.3 Ω	165 µV / Ω	4700 – 5250 Ω
Ref.	**Error**	**Power**	**Measurement Time**	**Processing Method**
[12]	0.022%	—	60 ms	10–12 MCU operations
[13]	0.18%	—	13 ms	10–12 MCU operations
[14]	0.29%	12 mW	15 ms + 3 discharge times	10–11 MCU operations
[15]	±0.28%	0.275 mW–2.03 μW	4.78 ms – 389.3 ms	Dual-slope timing sequence
[16]	0.11%	—	2 charge-discharge cycles	MCU timing measurement
[17]	—	86.41 μW @ 1 V	Adaptive conversion	8-bit UP/DN counter
[this work]	0.5%	1.32 mW @ 3.3 V	20 ms (depending on averaging window)	MCU relative timing measurement

In microcontroller (µC) –based systems, one can take advantage of the embedded ADC to reduce the external circuitry. Nevertheless, the typical power consumption of ADC embedded in commercial µC varies from sub-mW to few mW per conversion, depending on the operating frequency, building technology, and number of conversion bits [6 – 8].

Alternative methods can be divided into resistance-to-frequency converters, resistance-to-phase converters, and direct interface circuits (DIC), all of them relying on a digitization process carried out by a µC timer that performs a time-to-digital conversion [9]. The resistance-to-frequency and resistance-to-phase conversion methods often require additional passive and active elements like oscillators, operational amplifiers, voltage references, switches, or logic gates [8 – 11]. In the DIC-like cases, the resistive sensor is directly connected to the µC, being the measurement process based on the charge-discharge (C/D) timing of a capacitor, eventually involving some other resistors and multiple conversion cycles.

Table I lists representative resistive sensor interfacing circuits and highlights the reported respective characteristics and design trade-offs. The methods in [12] and [13] rely on resistance variable frequency oscillators, using a µC counter to obtain the oscillation frequency, i.e., the time taken to C/D a capacitor. [14] extends this principle to improve robustness against lead-wire resistance, but now, opposed to presetting the C/D duration, the proposed scheme configures and adapts the respective periods to perform the measurement. The self-heating-compensated digitizer described in [15] focuses on limiting sensor power dissipation through low excitation currents and current boosting, resorting to a dual-slope integrator and a comparator to generate a time sequence output signal, which is directly fed to the digital pin of a µC for time counting. The enhanced resistance-to-time interface in [16] targets faster conversion and lead-wire compensation using a reduced number of C/D cycles at the cost of a more complex external circuit involving a multivibrator, a set of switches, and diodes, to impose different C/D paths. Finally, in [17] the ADC-free adaptive interface uses a feedback-controlled counter/DAC structure to compensate baseline drift and reduce power in gas-sensor applications.

The next section describes the operating principle of the proposed approach. Section III presents the setups used to evaluate and compare its performance. The results are discussed in Section IV, and final conclusions are provided in Section V.

II. THE ΣΔ-BASED DIC APPROACH

Compared to the approaches listed in Table I, the scheme proposed here prioritizes lower hardware complexity and power consumption, rather than maximum resolution or lead-wire compensation. The scheme in Fig. 2 shows its principle of operation. The sensor R_S is connected between a reference voltage V_{Ref} and the µC digital input port P_i. Capacitor C connects between P_i and Ground. The auxiliary resistor R_F is connected between P_i and the digital output port P_o. During the measurement time T_m, several C charge and discharge cycles are performed. To achieve that, v_{Fed} is switched high (V_{DD}) when the capacitor voltage v_C reaches the V_{IL} voltage (lower limit) of P_i, forcing C to start a new charging cycle; when v_C reaches the P_i's V_{IH} voltage (upper limit) the output v_{Fed} is switched low (Ground) forcing C to start a new discharging cycle.

During each C/D cycle, a counter, operating at a higher frequency, is used to measure the respective time. Once the measurement time T_m is accomplished, the average value of voltage v_{Fed} (V_{Mes}) is calculated as in (1),

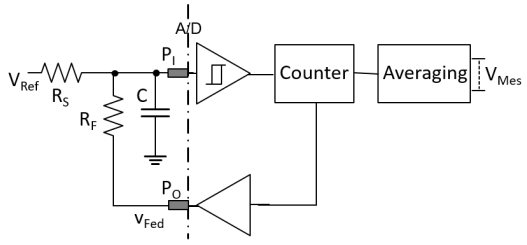


Figure 2. Block diagram of the direct-interface scheme proposed here.

$$< v_{Fed} >= V_{Mes} = V_{DD} \times \frac{N_c}{N_c + N_d} \quad (1)$$

where N_c is the total number of sampling periods counted during charging cycles, and N_d is the total number of counts during discharging cycles. This process is similar to that of a first-order ΣΔ A/D conversion process. In each instant, the capacitor current is given by the differential equation

$$C\frac{dV_C(t)}{dt} = \frac{V_{Ref} - v_C(t)}{R_S} + \frac{V_{Fed} - v_C(t)}{R_F} \quad (2)$$

whose solution is $v_c(t) = V_{Fin} + [Vc(0) - V_{Fin}]e^{-t/(R_{eq}C)}$, where R_{eq} is the parallel of R_S and R_F and V_{Fin} is either V_{DD} or *Ground*. The feedback loop operates by seeking to cancel the error that occurs between the instantaneous $v_c(t)$ voltage and the targeted voltage given by the voltage divider

$$V_\infty = \frac{V_{Ref}R_F + V_{Fed}R_S}{R_F + R_S}, \quad (3)$$

which is the midpoint swing of the input P_i logic comparator, typically ≈ V_{DD}/2 — this value has to be adjusted after calibration. Therefore, the V_{Ref} and R_{eq} values have to be selected so that the final capacitor voltage V_{Fin} is higher than the midpoint threshold in the charging cycle and lower in the discharging cycle. Once V_{Mes} is known, the sensor resistor is found from

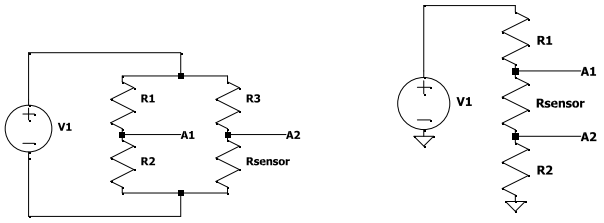
$$R_S = R_F \times \frac{V_{Ref} - \frac{V_{DD}}{2}}{\frac{V_{DD}}{2} - V_{Mes}} \quad (4)$$

Equation (4) shows that for higher detection sensitivity, V_{Ref} should be different from V_{DD}/2, either V_{DD} or *Ground* for maximum sensitivity.

III. Methods and Materials

Preliminary validation of the proposed DIC approach was carried out resorting to both simulation and experiments. Previous mechanical tests performed with a sensor embedded in an oral prosthesis subjected to forces from 0 to 400 N showed that the HAPTICA RS resistance varies approximately linearly between 4700 Ω and 5250 Ω.

For this validation, the sensor was emulated with resistors swept in steps of 50 Ω. For each resistance value, twelve measurements were obtained to evaluate the repeatability of the circuit response. All simulations and experiments were performed with a supply voltage of 3.3 V, matching the expected operating voltage of the microcontroller intended for future system integration.



(a) (b)

Figure 3. (a) Wheatstone bridge configuration. V1 = 3.3 V, R1 = R2 = R3 = 5.6 kΩ, and Rsensor = 4700 Ω : 5250 Ω. (b) Voltage-divider configuration. V1 = 3.3 V. Alternative scenarios: R1 = 0 and R2 = 4.7 kΩ; R1 = 1 kΩ, R2 = 3.9 kΩ; and R1 = 2.2 kΩ, R2 = 2.7 kΩ.

A. Simulation

All simulations were performed in LTSpice over a 20 ms analysis window. Although no noise source was included at this stage, this duration was adopted to provide enough time to accommodate capacitive transients. In particular, the extended window allowed the charging, discharging, and steady-state behavior of the capacitor to be observed without constraining the analysis to the initial response only. The same time interval was also adopted as a reference for the bench experiments, enabling a direct comparison between simulated and experimental measurements using equivalent observation windows.

A conventional Wheatstone bridge (Fig. 3.a) was used as a control reference. The bridge was implemented using fixed 5.6 kΩ resistors, together with the variable resistor representing the piezoresistive sensor. These resistance values implied a measurement $V_{A1} - V_{A2}$ offset but, on the other hand, allowed reducing the respective power dissipation.

The output voltage of the voltage-divider-based readout configuration (Fig. 3.b), is given by

$$|V_{out}| = |V_{A1} - V_{A2}| = V_{DD}\frac{R_{sensor}}{R_1 + R_{sensor} + R_2} \quad (5)$$

which depends on the sensing resistance and on the total series resistance, $R_1 + R_2$, rather than on the individual values of R_1 and R_2. Therefore, two configurations with the same total series resistance should produce the same differential output behavior. The removal of R_1 was also considered because it reduces the number of components and simplifies the readout structure. Three scenarios were simulated: 1) R_1 = 0, R_2 = 4.7 kΩ; 2) R_1 = 1 kΩ, R_2 = 3.9 kΩ; and 3) R_1 = 2.2 kΩ, R_2 = 2.7 kΩ. In the 2) and 3) cases the total series resistance is 4.9 kΩ.

As for the proposed DIC interface, the circuit shown in Fig. 4 was used, where R_1 = 920 Ω and C = 82 nF, while R_{sensor} was swept from 4700 Ω to 5250 Ω. Two V_{Ref} cases were evaluated, corresponding to V_1 = 3.3 V and V_1 = 0 V.

The circuit was modelled using a comparator with a threshold reference equal to approximately 0.6 V with a hysteresis window width of ≈ 15 mV.

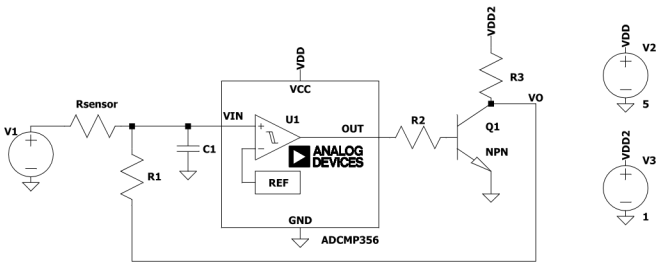


Figure 4. Scheme of the simulated ΣΔ DIC being proposed. $R_1 = 920\ \Omega, C_1 = 82\ nF, R_{sensor} = 4700\ \Omega : 5250\ \Omega$.

B. Experimental setup

For the experimental evaluation, the same nominal resistance range was considered. Instead of the real sensor, arrangements of discrete resistors, with 47 Ω increments, were used to simplify the practical resistor implementation. In all cases, an Arduino Due was used as the readout platform ($V_{DD} = 3.3\ V$). For the WB and VD configurations, the respective $V_{A1} - V_{A2}$ output voltages where captured with the 12-bit ADC built in the ATSAM3X8E ARM Cortex-M3 µC. After performing the WB and VD measurements, an additional test was performed to evaluate the effect of acquisition noise on the measured voltages. For both configurations, the Arduino Due sampled the output at 1 kHz, and the acquired values were processed using a sliding-window average. This was used to study the trade-off between measurement stability and acquisition duration, as shorter acquisition windows can reduce the active measurement time and, consequently, the average power consumption, provided that the output accuracy is not significantly affected. Three additional acquisition settings were tested. The first used a window length of 100 samples with a delay of 1 ms between updates. The second one used a window length of 20 samples with a delay of 5 ms. The third used a window length of 5 samples with a delay of 20 ms. These configurations preserve a comparable overall observation interval while changing the number of samples used in each averaging, allowing for a tradeoff among acquisition time, noise reduction, and measurement stability to be evaluated.

As for the proposed ΣΔ DIC interface case, only the average output voltage was acquired and used for comparison with the simulated response. The same 20 ms observation window was used. However, the first 500 µs were excluded from the average calculation to avoid the initial transient period of the feedback loop. The Arduino Due was configured to acquire the output at the highest sampling frequency achievable with the implemented code, i.e., $f_S \approx 118$ kHz. The current consumptions of all circuits were measured using a current sensor placed in the power supply.

Unlike the simulation model, where the switching threshold was defined by the fixed internal reference of the comparator, the Arduino Due digital input has an undefined transition region in which the logic state is unpredictable. For this reason, the external component values had to be adjusted experimentally. Three configurations were evaluated: $R_1 = 3047\ \Omega$ with $C_1 = 82$ nF and $V_1 = 0$ V; $R_1 = 2047\ \Omega$ with $C_1 = 82$ nF and $V_1 = 3.3$ V; and $R_1 = 2027\ \Omega$ with $C_1 = 82$ nF and $V_1 = 3.3$ V. These values were selected after studying the maximum usable value of R_1 for each input-source condition. The objective was to increase the output voltage variation while avoiding operation inside the undefined digital-input transition region. For $V_1 = 3.3$ V, the maximum evaluated value was $R_1 \cong 2044\ \Omega$, resulting in an output variation of about 385 mV. For $V_1 = 0$ V, the maximum evaluated value was $R_1 \cong 3056\ \Omega$, resulting in an output variation of about 460 mV.

TABLE II. SUMMARY OF SIMULATION RESULTS.

Configuration	Output Range [V]	Total Variation [mV]	Minimum Step [mV]
Wheatstone bridge	0.144 – 0.053	91	8
VD, R_2 = 4.7 kΩ	1.650 – 1.741	91	8
VD, R_1 =1 kΩ, R_2 = 3.9 kΩ	1.616 – 1.707	91	8
VD, R_1=2.2 kΩ, R_2=2.7 kΩ	1.616 – 1.707	91	8
ΣΔ DIC, V_1 = 3.3 V	0.053 – 0.109	56	5
ΣΔ DIC, V_1 = 0 V	0.690 – 0.678	12	1

IV. RESULTS AND DISCUSSION

As can be seen from Fig. 3 and 4, the number of external components used in each case are: 4 ($R_{sensor}, R_1, R_2, R_3$) for the WB, 2/3 ($R_{sensor}, R_2\ /\ R_1$) for the VD, and 3 ($R_{sensor}, R_1, C_1$) for ΣΔ-based DIC.

A. Simulation results

The output voltages obtained after simulation of the three topologies are summarized in Table II. In each case, the output range, total voltage variation, and minimum voltage step were extracted to compare the simulated readout behavior.

For the WB configuration, the output voltage decreased from approximately 144 mV at 4700 Ω to 53 mV at 5250 Ω, corresponding to a total variation of about 91 mV. The absolute voltage step between consecutive resistance values ranged approximately from 7.9 mV to 8.7 mV.

The VD configurations produced higher absolute output voltages, ranging from approximately 1.650 V to 1.741 V for the configuration with $R_2 = 4.7$ kΩ. The total voltage variation was approximately 91 mV, similar to the Wheatstone bridge. The two configurations with $R_1 + R_2 = 4.9$ kΩ produced equivalent results, confirming the prediction of (5). In all VD cases, the voltage step remained close to 8 mV.

For the proposed DIC topology, with $V_1 = 3.3$ V, the average output voltage increased from approximately 53 mV to 109.2 mV, resulting in a total variation of about 56 mV and a minimum voltage step close to 5 mV. With $V_1 = 0$ V, the

average output voltage decreased from approximately 690 mV to 678 mV, corresponding to a smaller total variation of about 12 mV and a minimum voltage step close to 1 mV.

B. Experimental results

The experimental results obtained with the Arduino Due readout are summarized in Table III. Overall, the WB and VD configurations followed the same trend observed within the simulations, presenting a nearly linear voltage variation over the emulated sensor resistance range.

TABLE III. EXPERIMENTAL RESULTS.

Configuration	Output Range [V]	Total Variation [mV]	Minimum Step [mV]
Wheatstone bridge	0.156–0.064	92	8
VD, R2 = 4.7 kΩ	1.662–1.753	91	7
VD, R_1=1 kΩ, R_2= 3.9 kΩ	1.627–1.719	92	7
VD, R_1=2.2 kΩ, R_2=2.7 kΩ	1.619–1.711	92	8
ΣΔ DIC, R_1=2047 Ω, V_1=3.3 V	0.556–0.679	123	11
ΣΔ DIC, R_1=2027 Ω, V_1=3.3 V	0.546–0.689	143	13
ΣΔ DIC, R_1=3047 Ω, V_1=0 V	2.284–2.193	91	7

In the WB case, the measured output voltage decreased from approximately 156 mV at 4700 Ω to 64 mV at 5250 Ω, that is, a total variation of about 92 mV, with a minimum absolute voltage step of ≈ 8 mV between consecutive resistance values. For the VD configurations, in the $R_2 = 4.7$ kΩ case, the measured voltage increased from ≈ 1.662 V to ≈ 1.753 V, resulting in a minimum voltage step of ≈ 7 mV. When R_1 was included, similar total variations were obtained, with output ranges of 1.627 – 1.719 V and 1.619 – 1.711 V, respectively. These results are consistent with the simulated behavior, where the differential voltage variation remained approximately unchanged for equivalent total series resistance values.

The influence of the acquisition time was also evaluated using the sliding-window settings described in Section III-B. Compared to the main acquisitions, the configurations using 100 samples with a 1 ms delay and 20 samples with a 5 ms delay showed a maximum deviation of ≈ ±1 mV. Shortening the window to 5 samples with a 20 ms time increased the deviation to ≈ ±5 mV. This confirms that shorter averaging windows increase the influence of acquisition noise, although the measured deviations remained within the same order of magnitude as the voltage steps reported for these configurations.

The experimental results followed the same general trend observed in the simulation of the WB and VD configurations. The measurements made with different acquisition windows showed that, as expected, the WB and VD outputs are improved after averaging. The cases using 100 samples and 20 samples produced deviations of ≈ ±1 mV, whereas reducing the window to 5 samples increased the deviation to ≈ ±5 mV. Considering that the voltage step between consecutive resistance values is close to 7 – 9 mV, very short averaging windows may hinder resistance resolution levels. Thus, the averaging window should be selected according to the required force resolution and power-consumption constraints.

Regarding the ΣΔ-based DIC, the experimentally extracted average output voltage showed a stronger dependence on the selected external component values. With $R_1 = 2047$ Ω, $C_1 = 82$ nF, and $V_1 = 3.3$ V, the average voltage increased from ≈ 0.556 V to ≈ 0.679 V, resulting in a total variation of about 123 mV. Using $R_1 = 2027$ Ω, $C_1 = 82$ nF, and $V_1 = 3.3$ V, the voltage increased from ≈ 0.546 V to ≈ 0.689 V, corresponding to a total variation of about 143 mV. In the configuration with $R_1 = 3047$ Ω, $C_1 = 82$ nF, and $V_1 = 0$ V, the average output voltage decreased from ≈ 2.284 V to ≈ 2.193 V, resulting in a total variation of about 91 mV. This is an aspect that deserves further clarification. Nevertheless, the targeted application may not require discrimination between so closely spaced resistance values.

TABLE IV. POWER SUPPLY CURRENT AND READOUT CIRCUIT POWER CONSUMPTION (V_{DD} =3.3V).

Configuration	Total Current [mA]	Circuit Power [mW]
WB	59.6	5.61
VD	58.8	2.97
ΣΔ DIC	58.3	1.32

Table IV summarizes the measured current consumption of each topology. Since the Arduino Due was used as the readout platform in all experimental tests, its baseline current (≈ 57.9 mA) was subtracted from the total measured current. The three topologies, WB, VD, and ΣΔ based DIC, presented currents of ≈ 1.7 mA, ≈ 0.9 mA, and ≈ 0.4 mA, respectively.

The current measurements further highlight the trade-off between readout simplicity and power consumption during measurement time. The higher current consumption of the WB and VD configurations is expected, since both architectures include continuous resistive paths between the supply and ground. In contrast, the ΣΔ-based DIC relies on feedback-based switching operation and avoids the need for an ADC block, which contributes to the lowest additional current consumption among the three topologies.

The total power consumption depends heavily on the readout platform. We used the Arduino Due because it is convenient and compatible with 3.3 V supply, but it is not designed for ultra-low-power use. Choosing a lower-power µC or a dedicated readout circuit in future versions will reduce power consumption and make the system more suitable for integration into an intraoral prosthesis.

When compared with the approaches summarized in Table I, the proposed circuit follows a different design priority. Several reported resistive sensor interfaces focus on high resolution, lead-wire compensation, or resistance-to-time conversion accuracy. These approaches are suitable for precision instrumentation, but they often require additional timing stages, multiple conversion cycles, or a larger set of external components. In contrast, the present work targets a highly constrained intraoral application, where compactness, low

power consumption, and reduced component count are primary requirements. Also, the expected temperature variation range is small and thus compensation for temperature variations is not a critical issue. The proposed ΣΔ-based DIC is not intended to outperform all previous methods in terms of linearity or resolution, but rather to provide a compact readout alternative for a piezoresistive force sensor embedded in an oral prosthesis.

V. CONCLUSION

This work presented and evaluated different readout circuits for a piezoresistive force sensor, intended for integration in an oral prosthesis. The main objective was to compare simple resistive sensor interfaces under the constraints of reduced size, low power consumption, and compatibility with compact embedded electronics.

A ΣΔ-based direct interface with a μC circuit has been proposed that outperforms the conventional Wheatstone bridge and voltage divider circuits in terms of power consumption and does not require the use of an ADC, while providing similar resolution. Compared to other direct interface circuits the solution being proposed presents a lower component count.

The undefined midpoint swing of the μC digital input port used as a 1-bit quantizer may introduce a non-linearity in the measurement. Nevertheless, the dense resistance sweep used in this study represents a demanding characterization condition, and broader calibration intervals may be sufficient for the targeted intraoral force-monitoring application.

Ongoing work is focusing on improving the precision and monotonicity of the topology being proposed, as well as on replacing the prototyping platform with a lower-power embedded implementation suitable for intraoral prosthetic integration.

REFERENCES

[1] S. E. Johnsen, Human Periodontal Mechanoreceptors: Functional Properties and Role in Jaw Motor Control. Ph.D. dissertation, Karolinska Institutet, Sweden, 2005.

[2] M. Trulsson, K. K. van der Bilt, A. Carlsson, and G. Svensson, "From brain to bridge: Masticatory function and dental implants," *Journal of Oral Rehabilitation*, vol. 39, no. 11, pp. 858–877, 2012, doi: 10.1111/j.1365-2842.2012.02340.

[3] Elena Sanz, Solange Vasquez-Ramos, Seyed Ali Mosaddad, Marta Revilla-León, Dr Miguel Gómez-Polo, "Prosthesis wear and complications in various materials used for fixed implant-supported prostheses: A systematic review". The Journal of Prosthetic Dentistry, ISSN 0022-3913, https://doi.org/10.1016/j.prosdent.2026.03.044, 2026.

[4] A. Watanabe, Y. Kanemura, K. Tanabe, and T. Fujisawa, "Effect of electromyogram biofeedback on daytime clenching behavior in subjects with masticatory muscle pain," *Journal of Oral Rehabilitation*, vol. 39, no. 3, pp. 200–207, 2012, DOI: 10.1016/j.jpor.2010.09.003.

[5] R. Casanella and R. Pallas-Areny, "On the design of low-power signal conditioners for resistive sensors," in Proc. 19th IMEKO World Congr. Fundam. Appl. Metrol., ISBN: 978-1-61567-593-7, pp. 787–791, 2009.

[6] F. Reverter and M. Gasulla, "Experimental Characterization of the Energy Consumption of ADC Embedded into Microcontrollers Operating in Low Power," IEEE Int. Instrumentation and Measurement Tech. Conf. (I2MTC), pp. 1-5, 2019, DOI: 10.1109/I2MTC.2019.8826815.

[7] J.-N. Kim, S.-K. Kwon, B.-C. Park, and H.-J. Kim, "A low-power portable gas sensor system with adaptive ROIC and Wi-Fi communication for biomedical applications," Chemosensors 2025, 13, 303. https://doi.org/10.3390/chemosensors13080303.

[8] A. R. Mohamed and M. A. Al Absi, "Readout Architectures for Resistive Sensors: Metrics, Comparison, and Future Directions," Arab J. Sci. Eng., vol. 51, pp. 11709–11728, 2026, https://doi.org/10.1007/s13369-026-11132-1.

[9] D. M. Wilson, "Electronic Interface Circuits for Resistance-Based Sensors," in *IEEE Sensors Journal*, vol. 22, no. 11, pp. 10223-10234, 1 June1, 2022, DOI: 10.1109/JSEN.2021.3124766.

[10] F. Reverter, A. C. Sreekantan and B. George, "Circuits for the Measurement of Remote Resistive Sensors: A Review," IEEE Trans. Instrum. Meas., vol. 74, pp. 1–13, 2025. DOI 10.1109/TIM.2025.3542136

[11] G. M. Puentes-Conde et al., "Direct Interface Circuits for Resistive, Capacitive, and Inductive Sensors: A Review," Electronics, vol. 14, 2393, 2025. https://doi.org/10.3390/electronics14122393.

[12] V. N. Philip, "Direct microcontroller interface based digital readout circuit for single-element resistive sensors," *IEEE Int. Conf. on Power, Control, Signals and Instrumentation Engineering (ICPCSI)*, Chennai, India, 2017, pp. 2480-2483, doi: 10.1109/ICPCSI.2017.8392163.

[13] H. Kapoor, K. Elangovan, and A. C. Sreekantan, "Charge-discharge-based digitizer for resistive displacement sensor," in Proc. ICST, pp. 1–5, 2023.https://api.semanticscholar.org/CorpusID:49334957

[14] K. Elangovan, A. Antony, and A. C. Sreekantan, "Simplified digitizing interface-architectures for three-wire connected resistive sensors," IEEE Trans. Instrum. Meas., vol. 71, pp. 1–9, 2022. doi 10.1109/TIM.2021.3136176.

[15] A. D. Joshi, S. Sohail, G. Singh, N. P. Kumar and T. Islam, "An Efficient Digitizer for Self-Heating Compensation in Resistive Sensor," *Int. Conf. on Recent Advances in Electrical, Electronics & Digital Healthcare Technologies (REEDCON)*, New Delhi, India, 2023, pp. 454-459, doi: 10.1109/REEDCON57544.2023.10150537.

[16] R. Anandanatarajan, U. Mangalanathan, and U. Gandhi, "Enhanced microcontroller interface of resistive sensors through resistance-to-time converter," IEEE Trans. Instrum. Meas., vol. 69, no. 6, pp. 2698–2706, 2020, doi: 10.1109/TIM.2019.2928348.

[17] C. -L. Chang, S. -W. Chiu and K. -T. Tang, "An ADC-free adaptive interface circuit of resistive sensor for electronic nose system," *2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, Osaka, Japan, 2013, pp. 2012-2015, doi: 10.1109/EMBC.2013.6609925.

A Dual-Head Model for Host based Intrusion Detection on System-Call traces

Matthias Dippold, Sofia Maragkou, Matthias Wess, Thilo Sauter
Institute of Computer Technology (ICT)
TU Wien
Vienna, Austria
mat.dippold@gmail.com, sofia.maragkou@tuwien.ac.at, matthias.wess@tuwien.ac.at, thilo.sauter@tuwien.ac.at

Grigorios Chrysos, Sotiris Ioannidis
Dept. of Electrical and Computer Engineering (ECE)
Technical University of Crete
Chania, Greece
gxrysos@tuc.gr, sioannidis@tuc.gr

Abstract— **Host-based intrusion detection systems (HIDS) need to detect both unseen exploits from day one and recurring known attack families. Maintaining separate anomaly-detection and supervised-classification frameworks increases the computational and operational footprint — a cost that only a few embedded or edge devices can carry. To satisfy both requirements without increasing deployment overhead, we propose a single dual-head Convolutional Neural Network (CNN) operating over a sliding window of embedded system-call tokens. Head 1 is an autoregressive next-token prediction module that emits sequence-level anomaly scores derived from token-wise negative log-likelihood (NLL) estimates, while Head 2 is a supervised attack-classification module using a shared latent representation. The two heads jointly support concurrent anomaly detection and supervised attack classification within a single model.**
The proposed framework is evaluated using the ADFA-LD Linux system-call benchmark [1]. The proposed framework achieves mean AUROC scores of 0.916 in the unsupervised setting and 0.979 in the supervised setting, outperforming the baselines reported in [2] under the reshuffled evaluation protocol in both settings. These results demonstrate that unified sequential representation learning can simultaneously support both zero-day anomaly detection and supervised attack classification refinement within a single operational HIDS framework, thereby reducing the operational complexity of practical HIDS deployments.

Keywords-host-based intrusion detection; anomaly detection; deep learning; convolutional neural networks; zero-day attacks

I. INTRODUCTION

For decades, the growth of network traffic volume has been accompanied by a corresponding increase in cyberattacks, making Intrusion Detection Systems (IDS) the standard mechanism for detecting malicious activity before it reaches downstream applications. Two principal detection frameworks dominate intrusion detection research and deployment. Signature-based detection frameworks match incoming events against curated databases of malicious patterns. While highly effective against previously observed attacks, such systems are inherently limited to known threats, leaving zero-day exploits and modified variants undetected until new signatures are deployed [1]. Anomaly-based detection frameworks address this issue by modelling normal system behaviour from historical benign data and flagging deviations as potential threats, enabling zero-day coverage without prior signatures [3], [4]. Machine Learning (ML) techniques have become the dominant implementation strategy for both paradigms [5], [6].

Practical HIDS operate across two distinct phases of the threat lifecycle. Supervised approaches require representative labelled samples from all target attack classes during training. However, systems are typically deployed under cold-start conditions, where benign traces are abundant but labelled attack samples are scarce and become available only gradually as detected incidents are analysed retrospectively [2], [5]. Anomaly-based methods sidestep this by training on benign data alone, but at the cost of elevated false-positive rates that increase operational overhead [7].

System-call traces are inherently sequential and malicious behaviour often exhibits temporally ordered execution patterns and contextual dependencies between events. Consequently, effective intrusion detection depends not only on statistical syscall composition, but also on preserving temporal context within execution traces. While recurrent architectures have traditionally dominated sequential intrusion detection research, convolutional sequence models provide an efficient alternative due to their parallelizable structure, reduced computational complexity and ability to capture local temporal patterns within syscall sequences. Moreover, convolutional sequence models avoid the sequential training bottlenecks of recurrent architectures and they can offer improved efficiency compared to attention-based models in medium-length syscall contexts.

To address these challenges, we propose a dual-head CNN architecture that combines anomaly-based and supervised intrusion detection, both operating on a shared intermediate trace representation. The model is deployable from day zero using only benign traces and it progressively incorporates labeled attack families as they become available, without requiring architectural redesign or sacrificing zero-day detection capability. The proposed approach is evaluated on the ADFA-LD benchmark, which contains attack traces specifically designed to resist trivial detection [1].

Manuscript received May 25, 2026; revised July 22, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.143

To the best of our knowledge, this is the first HIDS framework explicitly designed to unify cold-start anomaly detection and progressively supervised attack classification within a shared sequential learning architecture. The main contributions of this work are summarized as follows:

- This work introduces *Conv1D-based CNN architecture* for system-call intrusion detection that jointly models anomaly detection and supervised attack classification within a shared latent space, integrating autoregressive next-token prediction with discriminative learning.
- A lifecycle-aligned training strategy is developed to support cold-start deployment using only benign traces, while progressively incorporating labelled attack families during fine-tuning without compromising zero-day detection performance, as measured by AUROC.
- Experimental results on ADFA-LD reveal consistent improvements, achieving superior AUROC and F1 performance vs. strong prior baselines in both unsupervised and supervised settings.
- Last, this work provides empirical evidence that system-call ordering and temporal context are critical factors for effective intrusion detection, highlighting the importance of sequential structure beyond static syscall statistics.

The rest of this paper is organized as follows: Section II reviews related work; Section III describes the methodology; Section IV presents the experimental setup; Section V reports the results; Section VI discusses findings and limitations; and Section VII concludes.

II. Related Work

Related host-based intrusion detection research has explored anomaly detection, supervised classification and hybrid learning approaches, often under different assumptions regarding sequence modeling, supervision and deployment lifecycle adaptation.

A. Benchmarks and Evaluation Protocols

System-call HIDS research relies on a small set of public benchmarks, including UNM, DARPA, ADFA-LD [1], and LID-DS [8], each capturing only a limited subset of benign and malicious behaviour observed in production environments [5], and are often used under incompatible train/test protocols that complicate direct comparison across studies [9]. Several recent supervised approaches report very high scores on the original ADFA-LD split, for example He et al. report F1 = 0.972 and AUROC = 0.992 [10], and Shamim et al. report F1 = 0.990 and AUROC = 0.992 [11]. Nevertheless, Kim et al. show that the original Train and Validation pools are not drawn from the same distribution, so models trained on the released Train split generalize poorly and the original partition is unsuitable for deep-learning evaluation [2]. To address this, Kim et al. propose a reshuffled protocol that merges the released Train and Validation benign-sequence pools before re-partitioning into train/validation/test, making the split suitable for representation learning. More recently, Landauer introduced a reproducible 1%-train / 99 %-test ADFA-LD benchmarking pipeline for consistent comparison across methods [9], while LID-DS extended syscall-based evaluation with richer execution metadata, including thread information, syscall arguments and return values [8].

B. Representation Strategies for System-Call Traces

System-call traces can be represented as discrete, variable-length token sequences, and prior HIDS approaches differ primarily in how these sequences are encoded. Recent deep-learning approaches increasingly rely on dense sequential embeddings; however, most of them continue to separate anomaly detection and supervised classification into independent deployment stages. Embedding-based pipelines map each trace to a fixed vector that retains order. Kim et al. [2] represent ADFA-LD traces using a fixed embedding, i.e., Doc2Vec [12], RNN autoencoders and denoising RNN autoencoders, while anomaly detection is performed by measuring deviation from benign embeddings. Once labeled attacks become available, supervised classifiers are trained on the same latent vectors. Similarly, Gungor et al. [13] combine DeepSVDD-trained CNN embeddings with isolation-based novelty detection on LID-DS [8]. Despite substantial differences in representation strategy and architectural complexity, most prior approaches continue to treat anomaly detection and supervised attack classification as separate operational stages. In many embedding-based and deep sequential pipelines, the anomaly detector is trained exclusively on benign traces during deployment initialization, while supervised classifiers are introduced later as independent models once labeled attack families become available. Consequently, integrating supervised knowledge often replaces the original anomaly detector rather than extending it within a unified lifecycle-aware detection framework.

C. Hybrid and Lifecycle-Aware Intrusion Detection

Recent intrusion detection research has increasingly explored hybrid architectures that combine multiple detection objectives, representation strategies, or decision mechanisms within a shared framework. Li et al. propose a dual-head network with a shared parallel CNN–BiLSTM backbone feeding separate binary attack-detection and multiclass attack-type classification heads that are trained jointly under supervision, with the detection head gating the type prediction at decision time in network intrusion detection [14]. Similarly, Kamal et al. employ a serial Autoencoder–Dense–Transformer pipeline with adaptive resampling and class weighting to improve supervised detection robustness under imbalanced traffic distributions [15]. Consequently, these methods remain unsuitable for cold-start HIDS deployment where labeled attack traces are initially unavailable.

Other works seek to improve robustness against previously unseen attacks by combining discriminative and generative anomaly signals. Cao et al. combine OpenMax-based open-set recognition with variational autoencoder reconstruction scoring to identify samples that deviate from the training distribution [16]. On the host side, SeHIDS [17] adjusts model size as system

behaviour evolves over time, but still trains a separate signature-based and anomaly-based detector for each deployment. Nevertheless, these approaches continue to maintain separate anomaly-based and signature-based detection components rather than integrating both objectives within a unified learning framework. Collectively, prior hybrid and adaptive intrusion detection systems continue to treat anomaly detection and supervised attack classification as distinct operational stages. As labeled attack families become available, supervised components are typically introduced as independent replacement models rather than extensions of the original anomaly detector. Consequently, existing hybrid intrusion detection systems remain only partially aligned with the operational lifecycle requirements of real-world HIDS deployment.

III. Methodology

The proposed framework combines anomaly detection and supervised attack classification over sequential system-call traces within a dual-head architecture, supporting both cold-start and progressively supervised phases of a host-based IDS deployment. Under the practical lifecycle of HIDS, the system must (i) identify previously unseen attacks from benign traces alone, (ii) incorporate labeled attack knowledge to improve precision on recurring families, and (iii) preserve anomaly-based detection capability after supervised fine-tuning.

A. Dataset

Following Landauer [9], the ADFA-LD benchmark [1] was selected for evaluation of the proposed framework. ADFA-LD contains 175 distinct syscall identifiers distributed across 833 benign training traces, 4372 benign validation traces, and 746 attack traces spanning six attack families: Adduser, Hydra-FTP, Hydra-SSH, Java-Meterpreter, Meterpreter and Web-Shell. Among the syscall-based intrusion datasets surveyed by Landauer, ADFA-LD exhibits the highest normalized N-gram entropy and does not expose trivial detection shortcuts based on trace length or previously unseen syscall tokens.

B. System Architecture

The proposed framework consists of a shared sequential convolutional encoder feeding two complementary detection heads operating on a common latent representation z, as illustrated in Fig. 1: an autoregressive next-token decoder producing a Negative Log-Likelihood (NLL) anomaly score, and a supervised binary classifier for known attack families. The two scores are combined through a score-fusion stage into the deployment-time detection score p_{fusion}. The input representation and core components are described below.

1) Input Representation: Each system-call trace is represented as a discrete sequence over a vocabulary of 176 token identifiers comprising the 175 distinct ADFA-LD syscall IDs together with a reserved PAD (padding) symbol at index 0. To convert variable-length execution traces into fixed-size inputs suitable for convolutional processing, a sliding window of length W and stride $S = 1$ is applied across each sequence. For each prediction position t, the input window is defined as

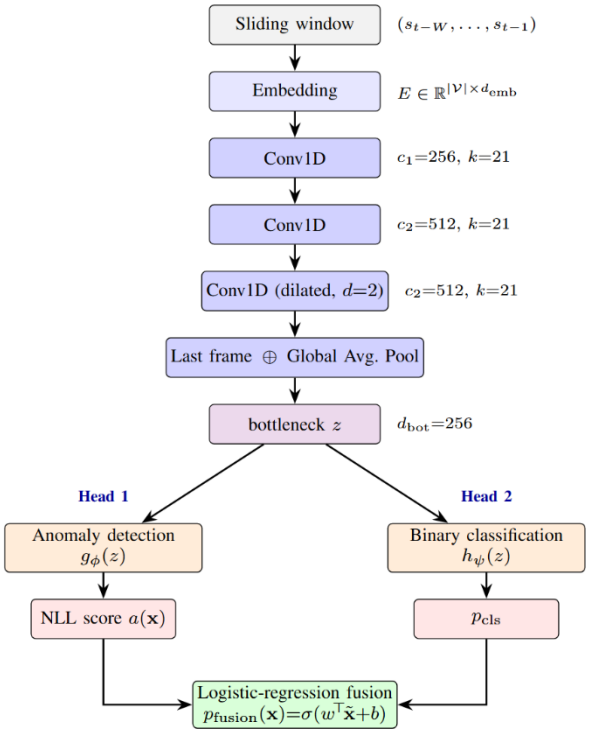


Figure 1. Dual-head CNN over a sliding window of system-call token embeddings. Head 1 (left) is a next-token-prediction decoder providing an NLL anomaly score; Head 2 (right) is a binary classifier on the shared bottleneck. The two scores are combined into a single score p_{fusion} at deployment.

$\boldsymbol{x}_t = (s_{t-W}, \ldots, s_{t-1})$, where $\boldsymbol{x}_t$ is an integer vector of length W containing the syscall tokens immediately preceding the target position. Each token window is mapped through a learned embedding matrix $E \in \mathbb{R}^{|\mathcal{V}| \times d_{emb}}$ producing an embedded representation $\boldsymbol{x}_t$ to $X_t \in \mathbb{R}^{W \times d_{emb}}$. The resulting embedding sequence forms a one-dimensional signal of length W with d_{emb} channels, which is processed by the Conv1D encoder.

2) Shared CNN Encoder: The shared encoder $f_\theta: \mathbb{R}^{W \times d_{emb}} \to \mathbb{R}^{d_{bot}}$ consists of three Conv1D blocks of kernel size k, where the first two blocks use c_1 and c_2 output channels respectively, while a third dilated Conv1D block with dilation $d = 2$ enlarges the temporal receptive field. Each convolutional block is followed by ReLU activation and batch normalization. The final temporal representation is obtained by concatenating the last feature frame with its global-average pooled representation, followed by layer normalization and linear projection into a bottleneck embedding $z \in \mathbb{R}^{d_{bot}}$. Hyperparameters are selected based on the sensitivity analysis described in Sec. IV-C. The final configuration uses $W = 32$, $S = 1$, $c_1 = 256$, $c_2 = 512$, $k = 21$, $d_{bot} = 256$, $d_{emb} = 256$, corresponding to an approximately 10 M-parameter backbone.

3) Head 1 – Autoregressive Anomaly Detection: The anomaly-detection head $g_\phi: \mathbb{R}^{d_{bot}} \to \mathbb{R}^{|\mathcal{V}|}$ performs autoregressive next-token prediction over syscall sequences.

Given an input window $\boldsymbol{x}_t = (s_{t-W}, \ldots, s_{t-1})$ the decoder predicts the subsequent syscall token s_t. Since the prediction target lies outside the observed window, no causal masking is required within the encoder. The anomaly loss is the expected per-window Negative Log-Likelihood [19], $\mathcal{L}_A = \mathbb{E}_{(\boldsymbol{x}_t, s_t)}[NLL(s_t \mid \boldsymbol{x}_t)]$, computed over training windows extracted exclusively from benign traces. During inference, per-window NLL values are aggregated into sequence-level anomaly scores using mean, maximum and p95 statistics per sequence for calibration and evaluation.

4) Head 2 – Supervised Binary Classification: The supervised classifier $h_\psi: \mathbb{R}^{d_{bot}} \to [0,1]$ consists of a single-layer MLP with sigmoid activation operating on the shared bottleneck representation. The supervised loss $\mathcal{L}_B$ is formulated as binary cross-entropy over (window, attack-flag) pairs. To reduce class imbalance during fine-tuning, training batches are sampled using a balanced loader that matches the number of benign and labeled attack windows.

5) Score Fusion: During inference, the sequence-level anomaly score $a(\boldsymbol{x}) = \overline{NLL}$ and the classifier probability $p_{cls} = h_\psi(f_\theta(\boldsymbol{x}))$ are combined through a logistic-regression fusion layer trained on the validation set. The raw fusion input vector $[log(1 + a(\boldsymbol{x})), p_{cls}]$ is first standardised and subsequently processed by an L_2-regularised, class-balanced logistic regression model, yielding the deployment score $p_{fusion}(\boldsymbol{x}) = \sigma(w^\top \tilde{\boldsymbol{x}} + b)$. The anomaly feature is log-compressed to reduce the impact of absolute-scale drift between validation and test partitions. The final decision threshold τ is selected on the validation set by maximising F1-score.

C. Two-Stage Training Strategy

Stage A corresponds to unsupervised cold-start deployment and trains the encoder together with the autoregressive anomaly-detection head exclusively on benign windows for 80 epochs using the Adam optimizer [20] at a peak learning rate of 5×10^{-4}. After each epoch, sequence-level NLL is evaluated on the validation set (benign sequences together with all attack sequences) and the checkpoint achieving the highest validation NLL-AUROC is retained. Including unknown-family attacks in this selection signal slightly inflates the reported zero-day sensitivity, since the held-out families are seen during checkpoint choice; downstream calibration of the fusion layer and decision threshold τ (Sec. III-B5) uses only the known subset, matching the operational lifecycle.

Stage B performs 30 epochs of supervised fine-tuning once labeled attack families become available. The encoder parameters use a backbone learning rate of 1×10^{-5}, ten times lower than the classifier head (1×10^{-4}). The two heads are optimized jointly under $\mathcal{L} = \mathrm{BCE}(h_\psi(z), y) + \lambda_{\mathrm{recon}} \cdot \mathbb{E}_{\mathbf{x} \sim \mathrm{normals}}[\mathrm{NLL}(g_\phi(z), \mathbf{x})]$ with $\lambda_{\mathrm{recon}} = 0.5$. The reconstruction term is computed only on benign windows, so the anomaly head never learns to model attacks. Each batch contains an equal number of benign and labeled-attack windows. The checkpoint with the highest composite score cls-AUROC + 0.5 · NLL-AUROC is kept, subject to a hard floor of NLL-AUROC ≥ 0.75. Stage B is activated only when labeled families exist; otherwise the system runs under Stage A alone.

IV. Experimental Setup

A. Evaluation Metrics

AUROC serves as our primary metric, being threshold-independent and robust to class imbalance [18]. The primary AUROC is computed in pooled binary form, scoring all attack traces against all benign traces, and is the most realistic real-world metric because a deployed detector faces novel zero-day attacks that cannot be attributed to any known family and must simply be flagged as anomalous; the per-attack-family breakdowns across the six ADFA-LD families are also reported for comparability with prior work [2]. F1-score, precision, and recall are additionally reported at a validation-derived threshold τ. Because benign NLL scores drift slightly between validation and test, a threshold tuned on the validation set is not perfectly calibrated on the test set. AUROC, which depends only on score ordering, is unaffected by this shift. Preservation of zero-day detection after finetuning with three out of six attack classes is assessed by comparing unsupervised AUROC before and after fine-tuning.

B. Evaluation Protocols

Because the released Train/Validation partition is unsuitable for deep-learning, as discussed in Sec. II, two complementary protocols are adopted in this work.

1) Landauer 1%-train split: This split of the ADFA-LD dataset was originally intended for classical sequence-based detectors. Under this setup, 1% of benign traces are used for training, while the remaining benign traces together with the full attack pool are reserved for testing [9]. We adopt this protocol with minimalistic training pool to benchmark against classic detectors. While [9] reports only F1 (not AUROC) under this protocol, our corresponding comparison in this work is likewise F1-based (Sec. V-B).

2) Kim-Style Reshuffled Protocol: The primary evaluation protocol follows the reshuffled deep-learning-oriented setup proposed by Kim et al. [2]. The original Train and Validation benign pools are first merged (5205 traces), randomly reshuffled at sequence level, and partitioned into a fixed 70/15/15 train / validation / test split. This produces 3643 benign training traces, 781 validation traces, and 781 test traces, substantially increasing the effective training pool compared to the original 833-trace Train partition while remaining within the upper range of the training sizes swept by Kim et al. Attack traces are distributed proportionally across the same partitions, resulting in 522/112/112 attack sequences for train/validation/test respectively. During the unsupervised anomaly detection stage, attack traces are excluded entirely from training. Labeled attack families are incorporated only during the supervised fine-tuning stage described in Sec. III-C. This reshuffled protocol enables stable deep-learning evaluation, supports lifecycle-aligned supervised refinement, and facilitates direct comparison with prior deep-learning

baselines evaluated under equivalent experimental conditions. The matching deep-learning baselines under this protocol are the RNN denoising autoencoder of [2] combined with Isolation Forest (unsupervised) and a Random Forest (supervised). Numeric values are reported in Sec. IV-D.

TABLE I. ARCHITECTURAL SENSITIVITY ANALYSIS (STAGE A VALIDATION AUROC). EACH ROW PRESENTS A SINGLE ARCHITECTURAL PARAMETER. THE PARAMETER COUNT IS HELD CONSTANT WITHIN THE WINDOW-SIZE SWEEP AND INCREASES MONOTONICALLY ELSEWHERE.

Parameter (range)	Min → Peak → Max	AUROC range
Window W	1 …32 …256	0.734 / 0.863 / 0.838
Kernel k	3 …31	0.825 / 0.858
Channels c_1/c_2	8/16 …256/512	0.787 / 0.881
Bottleneck d_{bot}	4 …512	0.810 / 0.851
Embedding d_{emb}	4 …512	0.822 / 0.856

C. Architectural Sensitivity Analysis and Final Parameters

The architectural configuration adopted throughout this work is described in Sec. III-B2, and it is selected through a one-factor-at-a-time sensitivity analysis over five primary architectural parameters: window size W, kernel size k, convolutional channels c_1/c_2, bottleneck dimension d_{bot}, and embedding dimension d_{emb}. Each parameter is varied independently, while all remaining hyperparameters are held fixed. Model selection is based on unsupervised validation AUROC. Table I summarises the complete sensitivity-analysis results. The sensitivity analysis favours larger model capacity across all architectural dimensions except window size. Unlike the other parameters, W controls how many past syscall tokens are visible to the model per prediction step, so its sweep is directly tied to the sequential structure of ADFA-LD anomalies and is analysed further in Sec. V-A. Convolutional channel capacity produces the largest performance improvement, increasing validation AUROC from 0.787 to 0.881, while kernel performance saturates near $k = 21$ and the bottleneck representation saturates at $d_{\text{bot}} = 256$. In contrast, the embedding dimension has comparatively limited influence on performance, which is consistent with the relatively small syscall vocabulary of ADFA-LD (176 tokens). The final retained configuration (Sec. III-B2) is used consistently across all experiments, together with the two-stage training schedule of Sec. III-C.

D. Comparison Baselines

Baselines are grouped by the evaluation protocols defined in Sec. IV-B, so that head-to-head comparisons in Sec. V involve only methods sharing the same train/test partition. Under the Landauer 1 %-train protocol, the strongest baseline reported by Landauer [9] on ADFA-LD is an edit-distance detector with $F1 = 0.343$, which serves as the reference point for the cold-start comparison. Under the Kim-style reshuffled protocol, the deep-learning baselines of Kim et al. [2] are used: an RNN denoising autoencoder combined with Isolation Forest, reaching a macro-average per-attack AUROC of 0.825 (best 0.883, Java-Meterpreter) in the unsupervised setting, and the same encoder combined with a 100-tree Random Forest, reaching a macro-average per-attack AUROC of 0.967 (best 0.985, Hydra-FTP) under supervised training.

E. Evaluation Scenarios

Four evaluation scenarios are considered to emulate different stages of the operational HIDS lifecycle. **S1** reproduces the Landauer 1 %-train / 99 %-test protocol for direct comparison with classical sequence-based anomaly detectors. **S2** evaluates the unsupervised Stage A under the Kim-style reshuffled protocol with $|\mathcal{K}| = 0$ known attack families, across five random seeds. **S3** extends the same reshuffled protocol to the partially supervised Stage B configuration with $|\mathcal{K}| = 3$ randomly selected known attack families. **S4** corresponds to the fully supervised Stage B configuration with all six attack families available $|\mathcal{K}| = 6$, evaluated across five random seeds. Additionally, we investigate the sequential manifestation of anomalies in two experiments to answer whether ADFA-LD anomalies primarily reflect sequential structure or simple token-frequency artifacts [9]. First, the context window size W is varied while keeping the parameter count fixed, isolating temporal context from model capacity. Second, syscall traces are independently permuted per sequence, preserving token composition and sequence length while destroying temporal order and local n-gram structure. Both analyses are presented in Sec. V-A.

V. RESULTS

A. Sequential Manifestation of Anomalies

This section summarizes the experiments designed to evaluate the role of temporal context and syscall ordering in ADFA-LD anomaly detection.

1) Window-Size Sweep at Constant Parameter Count: The first experiment evaluates the contribution of sequential context by varying the window size W while maintaining a constant parameter count. Convolutional channel dimensions are adjusted accordingly so that all evaluated configurations contain exactly $325{,}425$ trainable parameters. As shown in Fig. 2, validation AUROC increases from 0.734 at $W = 1$ to 0.863 at $W = 32$, before gradually decreasing to 0.838 at $W = 256$. As the model capacity remains constant throughout the sweep, the observed improvement can be attributed to the availability of sequential temporal context rather than increased model expressiveness. These results provide initial evidence that anomaly detection on the ADFA-LD dataset depends on sequential structure rather than solely on marginal token statistics. The gradual degradation beyond $W = 32$ reflects signal dilution, where anomalous behavioural motifs occupy relatively short temporal spans and the larger windows introduce increasing amounts of benign context.

2) Per-Sequence Token Permutation: While the window-size sweep demonstrates that larger temporal context improves detection, it does not directly establish whether syscall ordering itself is necessary. The second probing experiment changes sequential structure by independently permuting syscall order within each trace while preserving the sequence length, the token composition, and the class labels. Consequently, only

temporal dependencies and local n-gram structure are removed, whereas all compositional statistics remain unchanged. The complete anomaly-detection pipeline is then retrained on the permuted dataset using identical architecture, hyperparameters,

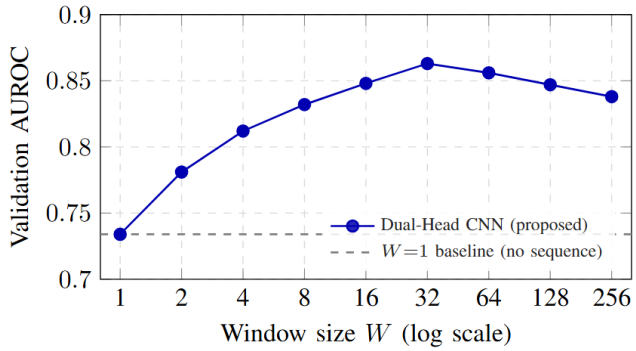


Figure 2. Stage A validation AUROC versus window size W with constant parameter count ($n_{params} = 325{,}425$). AUROC peaks at $W = 32$; the rise from $W = 1$ (unigram) is therefore attributable to sequential context rather than to additional model capacity.

TABLE II. PER-SEQUENCE SHUFFLE (STAGE A). TOKEN MULTISETS AND LENGTHS ARE UNCHANGED; ONLY POSITIONAL STRUCTURE IS REMOVED. AUROC DROPS SHARPLY ACROSS ALL ATTACK FAMILIES.

Metric	Ordered	Scrambled
Stage A best val AUROC	0.893	0.405
Test AUROC (fusion)	0.869	0.583
Final next-token cross-entropy	0.089	1.599

splits and random seeds. The performance degradation is reported in Table II: Stage A validation AUROC drops from 0.893 on the original ordered sequences to 0.405 on the scrambled sequences, while test AUROC decreases from 0.869 to 0.583. Similarly, the final next-token cross-entropy increases substantially from 0.089 to 1.599, indicating that little predictable structure remains once temporal order is removed. The normal/attack NLL separation further supports this interpretation. Under ordered traces, attacks produce higher mean NLL than benign traces (3.28 vs. 0.75), whereas under shuffled traces the separation reverses (2.44 vs. 2.70). This inversion suggests that non-sequential compositional statistics alone are insufficient for reliable anomaly discrimination and may actively mislead the detector once temporal structure is removed.

Combining the window-size sweep and the per-sequence permutation experiment results provide direct evidence that ADFA-LD anomalies manifest primarily through sequential execution patterns rather than token-frequency statistics. All remaining experiments use the same retained architectural configuration without further modification.

B. *Landauer 1% Train Split – Classical-Baseline Comparison*

Under the Landauer 1 %-train / 99 %-test protocol, the proposed framework is evaluated in Stage A mode using the NLL anomaly score without supervised fine-tuning. As summarized in Table III, the detector reaches $F1 = 0.403$ ($P = 0.379$, $R = 0.431$), improving over the strongest classical baseline of Landauer [9] (edit distance, $F1 = 0.343$) by approximately 6 F1 points under the same constrained 1 %

TABLE III. LANDAUER 1 %-TRAIN / 99 %-TEST UNSUPERVISED ADFA-LD PROTOCOL.[A]

Method	F1	P	R
Ours, $\lvert\mathcal{K}\rvert = 0$	**0.403**	0.379	0.431
Landauer 2024, edit distance	0.343	–	–

[A]Trained on only 1 % of benign traces; not directly comparable to the reshuffled-split tables. Landauer 2024 reports only F1 under this protocol.

TABLE IV. UNSUPERVISED STAGE A ($|\mathcal{K}| = 0$) ON THE [2] RESHUFFLED 70/15/15 SPLIT. ALL VALUES ARE AUROC. POOLED IS TAKEN OVER ALL ATTACKS VS. ALL NORMALS; AVERAGE AND BEST ARE COMPUTED OVER THE SIX ADFA-LD ATTACK FAMILIES.

Method	Pooled	Average	Best
Ours, $\lvert\mathcal{K}\rvert = 0$	0.916	**0.913**	**0.942**
Kim 2021, RNN-DAE + IF	–	0.825	0.883

training regime. This indicates that the learned sequential representation captures substantially richer behavioural structure than lightweight statistical sequence matching alone.

C. *Reshuffled Split – Stage A (Unsupervised)*

Under [2] reshuffled protocol, Stage A is evaluated across five independent random seeds. Since the Stage A training setup is based on fully unsupervised learning, the number of known attack families $|\mathcal{K}|$ has no effect on the training stage. The pooled binary test AUROC across the five runs is 0.916 ± 0.040; the corresponding mean $F1 = 0.636 \pm 0.125$ ($P = 0.531 \pm 0.166$, $R = 0.827 \pm 0.031$) at the validation-derived threshold. Kim et al. [2] do not report a pooled binary AUROC, so direct comparison on this metric is not possible. On the per-attack axes Kim does report, our framework reaches a macro-average per-attack AUROC of 0.913 and a best per-attack AUROC of 0.942 (Hydra-FTP), exceeding the corresponding [2] values of 0.825 average / 0.883 best on every individual attack family (Table IV).

D. *Reshuffled Split – Stage B Lifecycle Evaluation*

The Stage B fine-tuning phase progressively incorporates labelled attack families through the supervised classifier while preserving the anomaly-detection capability of the shared encoder, as described in Sec. III-C. Two deployment schemes are reported, emulating the lifecycle by progressively expanding the set of known attack families: $|\mathcal{K}| = 3$, where only three of the six ADFA-LD attack families are known and the remaining three families are zero-day at test time, and $|\mathcal{K}| = 6$, corresponding to a fully supervised deployment scenario. For the $|\mathcal{K}| = 3$ setting, each of five runs randomly selects three of the six known attack families, while the other three remain zero-day at test time. Under this configuration, the primary fusion score achieves mean $F1 = 0.654 \pm 0.097$ and mean pooled binary $AUROC = 0.940 \pm 0.024$. As $|\mathcal{K}| = 3$ is partially supervised, it is not directly comparable to either of [2] fully

unsupervised or fully supervised baselines. Crucially, supervised fine-tuning under $|\mathcal{K}| = 3$ does not erode the unsupervised anomaly path. To isolate this, the Stage B model's NLL score is re-evaluated on the test partition restricted to

TABLE V. SUPERVISED STAGE B AUROC ON THE [2] RESHUFFLED 70/15/15 SPLIT. POOLED IS TAKEN OVER ALL ATTACKS VS. ALL NORMALS; MACRO AND BEST ARE COMPUTED OVER THE SIX ADFA-LD ATTACK FAMILIES.

Method	Pooled	Macro	Best
Ours, $\|\mathcal{K}\| = 3$	0.940	–	–
Ours, $\|\mathcal{K}\| = 6$	**0.979**	**0.977**	**0.989**
Kim 2021, RNN-DAE + RF	–	0.967	0.985

benign sequences and the three attack families that were excluded from fine-tuning (the held-out zero-day families), and the resulting NLL-AUROC is compared against the Stage A model evaluated on the exact same subset. Averaged across runs, this Stage A → Stage B change is $\Delta = +0.001$, and every individual run stays within $[-0.001, +0.003]$ of its cold-start value. The safeguards described in Sec. III-C – recon-on-normals, reduced backbone learning rate, and the composite checkpoint criterion with NLL-AUROC floor – successfully prevent classifier-driven encoder drift. This shows that the dual-head design delivers *both* capabilities at once: the supervised F1/AUROC boost on the known families is gained *without* sacrificing the cold-start NLL detector on the unseen ones. This simultaneous-capability property directly satisfies requirement (iii) of Sec. III and is, to the authors' knowledge, the central novelty of the proposed architecture for the IDS lifecycle. The fully supervised $|\mathcal{K}| = 6$ configuration evaluated across five seeds achieves mean F1 $= 0.815 \pm 0.056$ and mean pooled binary AUROC $= 0.979 \pm 0.005$. Although [2] does not report a pooled binary AUROC, our $|\mathcal{K}| = 6$ model achieves a macro-averaged per-attack AUROC of 0.977, with a peak AUROC of 0.989 on Hydra-FTP. These results exceed the supervised RNN-DAE + Random Forest baseline reported in [2], which attains an average AUROC of 0.967 and a best AUROC of 0.985 across the individual attack families evaluated.

Table V reports the supervised lifecycle results, simulating the operational state of a HIDS after a substantial period of analysis. The $|\mathcal{K}| = 3$ row emulates a mid-lifecycle stage in which three families are still effectively zero-day, while $|\mathcal{K}| = 6$ corresponds to all currently catalogued ADFA-LD families being known. Fig. 3 visualizes the monotonic improvement in detection performance as more attack families become known.

E. Calibration and Distribution Drift

The training logs reveal a moderate distribution shift between the validation and test benign NLL distributions. The validation partition exhibits mean NLL $= 0.200$ (p95 $= 1.137$), while the test partition reaches mean $= 0.242$ (p95 $= 1.375$), corresponding to an approximately $1.21\times$ increase across both statistics. Although the observed drift is sufficiently small to preserve ranking-based evaluation through AUROC, it is large enough to affect fixed-threshold calibration. Consequently, validation-derived thresholds are used when reporting operational metrics such as F1-score, precision, and recall, rather than relying on absolute anomaly-score cut-offs.

VI. DISCUSSION

Head-to-head comparisons are limited to baselines evaluated under identical experimental protocols (Sec. III). The reshuffle

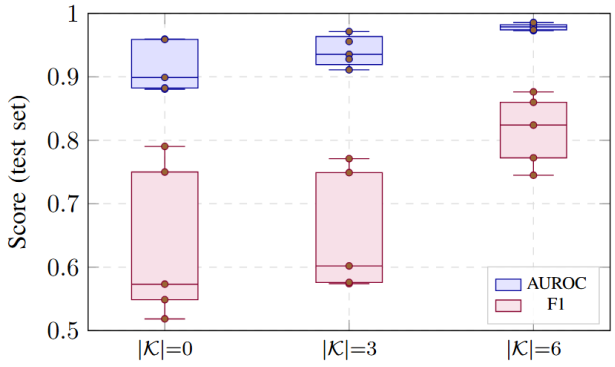


Figure 3. Lifecycle progression – the mean of both test AUROC (blue) and F1 (purple) increases monotonically as more attack families become known. Each scenario shows the distribution across $n = 5$ random seeds; for $|\mathcal{K}| = 3$, each run uses a random three-of-six subset of labelled families.

protocol of [2] seems to be the most appropriate setting for deep models since the original ADFA-LD Train and Validation normal sets are not drawn from the same distribution [2], so training and validating under the original split creates inconsistencies, making it difficult to tell whether poor scores reflect the model limitations or the partition effects.

Our dual-head architecture aligns with the IDS lifecycle: Stage A provides strong unsupervised detection under the matching protocol (Sec. V-C), and each labelled family incorporated in Stage B improves discrimination progressively (Table V). More importantly, the $|\mathcal{K}| = 3$ scenario shows that the supervised gain on known families is achieved *without* degrading the zero-day NLL detector on unseen families ($\Delta \approx 0$, Sec. V-D). This is supported by the reconstruction term in the Stage B loss, which helps preserve benign pattern representations and mitigate forgetting (Sec. III-C). To the best of our knowledge, this is the first system-call HIDS to demonstrate that a single shared encoder can carry both objectives simultaneously across the deployment lifecycle.

The ≈10 M trainable parameters of the encoder represent 99.5% of the total model parameters while the two detection heads + fusion layer, only add under 0.5% of the parameters. For identically sized encoders, two separate networks would nearly double both parameters and computational cost, so sharing the encoder roughly halves the deployment footprint. This efficiency could be optimized further on embedded and edge hosts.

Limitations and outlook: ADFA-LD provides a controlled and widely used benchmark for system-call intrusion detection, enabling reproducible lifecycle evaluation. We report results over five different seeds and random three-of-six subsets with stratified sampling, and use AUROC as the primary metric.

While this setting ensures comparability and stability of evaluation, it naturally serves as a stepping stone toward deployment-oriented regimes where decisions require explicit thresholding, e.g., via conformal or quantile-based calibration under sliding windows.

The proposed framework is lightweight and dataset-agnostic, relying only on the system-call alphabet and window size W, which enables straightforward transfer to benchmarks such as LID-DS [8]. This motivates future extension and evaluation to more complex and realistic operational settings, including multimodal intrusion detection where system calls are integrated with network and process-level signals within a unified representation space.

Finally, the dual-stage formulation highlights a further research direction in representation learning. While Stage B already leverages supervised signals for known attack families, a natural extension is to introduce explicit margin-aware objectives in latent space (e.g., contrastive or triplet learning over z). This extension would further shape inter-class geometry while preserving the reconstruction anchor, and is expected to improve both supervised discrimination and zero-day robustness within a unified framework.

VII. Conclusion

This work introduces a generic dual-head CNN framework for host-based intrusion detection that unifies the cold-start and progressively supervised phases of the HIDS lifecycle within a single architecture. The proposed solution provides zero-day detection from benign traces while seamlessly incorporating labelled attack families as they become available, without requiring any architectural modifications. Under the reshuffled protocol of Kim et al. [2], the framework achieves strong performance in the unsupervised cold-start setting (Stage A AUROC 0.916) and improves consistently as supervision is introduced, reaching an AUROC of 0.979. Across all settings, it outperforms strong RNN-DAE-based baselines under identical evaluation conditions, while preserving zero-day detection capability for unseen attack families. Beyond benchmark performance, this work demonstrates that anomaly-based and supervised intrusion detection can be unified within a single sequential representation, rather than treated as separate or sequentially replaced paradigms. A shared encoder is sufficient to support both cold-start zero-day detection and incremental supervised refinement within one model. Future directions include evaluation on more recent and complex benchmarks such as LID-DS [8], extension to multimodal fusion of system-call, process, and network event streams within a unified latent space, and adaptive threshold calibration mechanisms to improve robustness under operational drift in deployed systems.

References

[1] G. Creech and J. Hu, "Generation of a new IDS test dataset: Time to retire the KDD collection," in Proc. IEEE Wireless Commun. Netw. Conf. (WCNC), 2013, pp. 4487–4492.

[2] C. Kim, M. Jang, S. Seo, K. Park, and P. Kang, "Intrusion detection based on sequential information preserving log embedding methods and anomaly detection algorithms," IEEE Access, vol. 9, pp. 58 088–58 101, 2021.

[3] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in Proc. IEEE Symp. Security and Privacy, 1996, pp. 120–128.

[4] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in Proc. IEEE Symp. Security and Privacy, 1999, pp. 133–145.

[5] R. A. Bridges, T. R. Glass-Vanderlan, M. D. Iannacone, M. S. Vincent, and Q. Chen, "A survey of intrusion detection systems leveraging host data," ACM Computing Surveys, vol. 52, no. 6, pp. 128:1–128:35, 2019.

[6] S. A. Abdulkareem, C. H. Foh, M. Shojafar, F. Carrez, and K. Moessner, "Network intrusion detection: An IoT and non IoT-related survey," IEEE Access, vol. 12, pp. 144 608–144 636, 2024.

[7] A. Milenkoski, M. Vieira, S. Kounev, A. Avritzer, and B. D. Payne, "Evaluating computer intrusion detection systems: A survey of common practices," ACM Comput. Surv., vol. 48, no. 1, Sep. 2015. [Online]. Available: https://doi.org/10.1145/2808691

[8] M. Grimmer, T. Kaelble, F. Nirsberger, E. Schulze, T. Rucks, J. Hoffmann, and E. Rahm, "Lid-ds 2021," in CRITIS Conference Proceedings, 2021.

[9] M. Landauer, F. Skopik, and M. Wurzenberger, "A critical review of common log data sets used for evaluation of sequence-based anomaly detection techniques," Proc. ACM Softw. Eng., vol. 1, no. FSE, p. Article 61, 2024.

[10] J. He, C. Tang, W. Li, T. Li, L. Chen, and X. Lan, "BR-HIDF: An anti-sparsity and effective host intrusion detection framework based on multi-granularity feature extraction," IEEE Transactions on Information Forensics and Security, vol. 19, pp. 485–499, 2024.

[11] N. Shamim, M. Asim, A. I. Awad, and M. K. Khan, "Anomaly detection in Internet of Things system calls using a centroid-based vector-space model," IEEE Internet of Things Journal, vol. 12, no. 14, pp. 26 868–26 881, 2025.

[12] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in Proc. Int. Conf. Machine Learning (ICML), 2014, pp. 1188–1196.

[13] O. Gungor, I. Kale, J. Zhou, and T. Rosing, "Light-hids: A lightweight and effective machine learning-based framework for robust host intrusion detection," 2025. [Online]. Available: https://arxiv.org/abs/2509.13464

[14] T. Li, X. Zhang, H. Zhao, J. Xu, Y. Chang, and S. Yang, "A dual-head output network attack detection and classification approach for multi-energy systems," Frontiers in Energy Research, vol. 12, p. 1367199, 2024.

[15] H. Kamal and M. Mashaly, "Ae-dtnn: Autoencoder–dense–transformer neural network model for efficient anomaly-based intrusion detection systems," Machine Learning and Knowledge Extraction, vol. 7, no. 3, p. 78, 2025.

[16] Z. Cao, Z. Zhao, W. Shang, and S. Ai, "VAEMax: Open-set intrusion detection based on OpenMax and variational autoencoder," arXiv preprint arXiv:2403.04193, 2024.

[17] M. Baz, "SEHIDS: Self evolving host-based intrusion detection system for IoT networks," Sensors, vol. 22, no. 17, p. 6505, 2022.

[18] T. Fawcett, "An introduction to roc analysis," Pattern Recognition Letters, vol. 27, no. 8, pp. 861–874, 2006, rOC Analysis in Pattern Recognition. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016786550500303X

[19] I. Ring, John H., C. M. Van Oort, S. Durst, V. White, J. P. Near, and C. Skalka, "Methods for host-based intrusion detection with deep learning," Digital Threats: Research and Practice, vol. 2, no. 4, pp. 26:1–26:29, Oct. 2021.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in Proc. Int. Conf. Learning Representations (ICLR), 2015.

A Compiler-Aware Framework for Partitioned Neural Network Inference on FPGA DPUs

Federico Buccellato, Luca Mannini, Corrado De Sio
Politecnico di Torino
Turin, Italy
federico.buccellato@polito.it, s291065@studenti.polito.it, corrado.desio@polito.it

***Abstract*—The increasing adoption of Artificial Intelligence is driving the development of larger and more accurate neural networks. However, their high computational cost leads to significant inference latency, especially on resource-constrained embedded platforms such as FPGA-based systems. To address this issue, AMD introduced the Deep Learning Processing Unit, an accelerator integrated into the Vitis AI toolchain for efficient execution of quantized neural networks. Nevertheless, models with many parameters can be difficult to deploy on a single board due to latency or capacity constraints. In these scenarios, splitting a model into separately compilable fragments becomes useful for more flexible deployment. This process is not immediate within the Vitis AI flow. Our analysis shows that manually partitioning a network into independently compiled sub-networks can remove compiler optimizations. In particular, losing the global view of the quantized XIR graph can prevent the toolchain from preserving DPU mapping, moving accelerable operations to the CPU and causing severe performance degradation. Based on this observation, we analyze the Vitis AI compiler and propose an XIR-level splitting framework that generates independently compilable `.xmodel` fragments while preserving the context required for DPU mapping. The approach keeps the workflow high-level, without requiring advanced hardware design expertise or low-level design changes. Experimental results on CNN and ConvViT-based models show that naive splitting can introduce slowdowns up to ×249 on CNNs and ×542 on ConvViT variants. The proposed framework restores correct DPU mapping by addressing boundary-context loss and incomplete dependency collection, bringing latency back to the expected range for hardware-accelerated execution.**

***Index Terms*—FPGA, Distributed Inference, Deep Learning, DPU, Vitis AI, XIR**

I. Introduction

The growing adoption of Artificial Intelligence is driving the development of increasingly accurate and complex models, often based on deep neural networks with a large number of parameters [1]. Although these models achieve very high performance in many application domains, their computational cost represents a concrete limitation when they are deployed on embedded platforms. In these scenarios, strict latency constraints and limited available resources make the efficient execution of large models difficult.

FPGA-based architectures have emerged as a promising solution for edge inference, due to their flexibility, reconfigurability, and good trade-off between performance and energy efficiency. To exploit these features for neural network execution, FPGA-specific accelerators have been developed to optimize the most common operations in deep learning models, such as convolutions, matrix multiplications, and activation functions. In this direction, solutions such as the Deep Learning Processing Unit (DPU), introduced by AMD within the Vitis AI toolchain, represent an example of integration between hardware acceleration and high-level development flows, enabling a more accessible deployment compared to fully custom approaches.

Despite these advances, the continuous increase in model complexity often makes the use of a single FPGA board insufficient. Models with a large number of parameters may exceed the available resources or fail to meet the required latency constraints, making it necessary to distribute the execution across multiple devices. This requirement introduces new challenges related to model partitioning and the efficient management of distributed execution.

Traditionally, the problem of model parallelism and distribution has been widely studied in the context of high-performance platforms, especially GPUs and computing clusters. In these environments, techniques such as model parallelism and pipeline parallelism have been extensively explored and optimized, supported by mature software frameworks and dedicated infrastructures [2]. In contrast, in edge scenarios and on FPGA architectures, this problem is less explored and is often addressed through heterogeneous and poorly standardized solutions.

Existing approaches for distributed deployment on FPGAs are frequently based on custom implementations, which require manual design of inter-device connections, explicit management of data transfers, and deep knowledge of the underlying hardware [3]. This makes such solutions difficult to scale and less accessible in real application scenarios.

In this context, although accelerators such as the DPU have simplified FPGA deployment in single-device configurations, extending this approach to partitioned models remains problematic. In this work, we observe that a direct approach based on manually splitting the model into independently compiled sub-networks can lead to a loss of optimizations, since the compiler no longer has a global view of the network. This behavior can severely degrade performance, with slowdowns reaching up to ×542 compared to the optimized execution.

Starting from this analysis, we systematically study the behavior of the Vitis AI toolchain in model partitioning scenarios, showing that naive sub-network compilation can break the global context required for correct DPU mapping. To address

Manuscript received May 25, 2026; revised July 15, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.144

this issue, we propose an XIR-level splitting framework that partitions models at user-defined points while preserving the optimizations of the original compilation flow. The framework remains integrated in the high-level Vitis AI pipeline and produces fragments that can be coordinated through standard mechanisms such as TCP or SLURM. Experimental results show that the proposed approach recovers the optimizations lost by naive splitting and restores latency to the expected range for DPU-accelerated execution.

II. Related Works

With the increasing adoption of Artificial Intelligence, neural networks have reached larger sizes and higher complexity, increasing both memory requirements and the computational cost of inference. This growth has made it necessary to develop solutions able to support deeper and heavier models without compromising latency, throughput, and energy efficiency.

One of the most widely studied directions concerns the distribution of the computational workload across multiple resources. In high-performance systems, this problem has mainly been addressed on GPUs and computing clusters. Techniques such as data parallelism, model parallelism, and pipeline parallelism have been used to scale the execution of large neural networks by exploiting infrastructures with high computational capacity and high-bandwidth interconnections. Works such as DistBelief showed the possibility of executing large-scale models in distributed environments [2], while later approaches such as GPipe and PipeDream investigated pipeline parallelism for splitting deep networks into multiple execution stages [4], [5]. These solutions have played a central role in the scalability of neural models, but they are mainly designed for datacenter or HPC environments, which are very different from the embedded and edge scenarios considered in this work.

In the edge context, the problem has different characteristics. Embedded platforms must satisfy stricter constraints in terms of latency, energy consumption, available memory, and hardware cost. In this scenario, FPGAs are a particularly interesting platform for inference, due to their reconfigurability and their ability to implement specialized architectures for specific neural workloads. Several works have shown that FPGA accelerators can achieve good trade-offs between performance and energy efficiency for convolutional neural networks and quantized models [6]. However, obtaining these benefits often requires careful design of the architecture, memory management, and operation mapping on the device [7], [8].

To address the growth of modern models and the limited resources available on a single device, parallel approaches on edge and multi-FPGA platforms have also been studied. Some works have explored FPGA clusters for specific applications, such as recommendation system inference, using distributed pipelines and dedicated interconnections [3], [9]. Other multi-FPGA approaches, including OpenCL-based solutions, try to raise the abstraction level compared to traditional hardware design, but still require explicit management of partitioning, communication, and execution across devices. Overall, distributed execution on FPGAs often remains tied to custom solutions, which are difficult to generalize and not straightforward to integrate into a standard deployment pipeline.

Given the complexity of FPGA development and the limited generality of many existing custom solutions, several frameworks have been proposed in recent years to automate the generation of neural accelerators. In this context, FINN introduces a flow for quantized and binarized networks based on specialized dataflow architectures [10]. DNNWeaver generates synthesizable accelerators from high-level network descriptions [11], while fpgaConvNet addresses the automatic mapping of convolutional networks onto embedded FPGAs by optimizing the computational graph according to the available resources [12]. These toolflows highlight the importance of automation, but in many cases they still produce custom architectures or require expertise in hardware synthesis, accelerator configuration, and resource optimization.

To make FPGA acceleration more accessible also to users without specific hardware design expertise, the AMD ecosystem provides the Deep Learning Processing Unit integrated into the Vitis AI toolchain [13]–[16]. Compared to manually designed accelerators, the DPU allows a higher-level workflow, in which the model is quantized, compiled, and executed using tools already integrated in the toolchain. In this way, acceleration of quantized neural networks can be used without manually designing the accelerator datapath or directly modifying the hardware design.

However, the problem of applying this paradigm to modern large-scale models remains open, especially for networks with an increasing number of parameters, more complex structures, and latency constraints that are difficult to satisfy on a single board.

In these cases, a natural solution is to partition the neural network into multiple sub-models, so that each portion can again be managed within the Vitis AI flow for DPU execution. However, as shown in this work, manually splitting the network into independently compiled sub-networks can break the context used by the compiler and lead to the loss of the optimized mapping on the accelerator. This creates an intermediate issue between deployment and compilation. Splitting the model is not sufficient by itself; the split must preserve the graph information that allows the toolchain to maintain the original optimizations.

This work addresses this issue. Starting from a study of the Vitis AI toolchain behavior on partitioned models, we propose an XIR-level splitting framework that allows a network to be divided at user-defined points and generates independently compilable `.xmodel` fragments. Unlike approaches based on custom accelerators or specialized distributed runtimes, the framework keeps Vitis AI and the DPU as the starting point and operates on the intermediate graph to preserve the optimizations of the original compilation.

The framework reconstructs the quantization context at fragment boundaries and verifies the structural completeness of the extracted graph, including the dependencies required by branches, residual connections, and aggregation points. The evaluation was conducted on two model families, Parallel-

ExpertsNet and ConvViT, showing that naive splitting can cause slowdowns up to ×249 on CNN models and up to about ×542 on ConvViT models. In both cases, the framework restores effective DPU mapping and brings latency back to the expected range for hardware-accelerated execution. In this way, model partitioning remains compatible with a high-level workflow, without modifying the DPU or making low-level changes to the hardware design, and the generated fragments can be coordinated through standard mechanisms such as TCP or orchestration systems such as SLURM.

III. Technical Background

This section introduces the main tools and technological components used in this work. Since the proposed framework operates within the Vitis AI compilation and deployment flow, we describe the Deep Learning Processing Unit, the Vitis AI toolchain, the Xilinx Intermediate Representation, and SLURM as a workload manager for cluster environments.

A. Deep Learning Processing Unit

The AMD Deep Learning Processing Unit (DPU) is a dedicated hardware accelerator provided as a preconfigured overlay for FPGA and SoC platforms [13]. It is designed to accelerate neural network inference on AMD reconfigurable systems, especially for quantized models. By exploiting the parallelism of the FPGA fabric, the DPU reduces CPU involvement during the most computationally intensive stages of inference.

The DPU uses a parallel execution model based on specialized pipelines for common neural network operations, including convolutions, pooling, activation functions, and matrix multiplications. An internal control engine manages on-chip dataflow and coordinates computation through an optimized memory hierarchy. Intermediate results can be stored locally, while weights and larger data structures are placed in external DDR memory. This organization enables high throughput and low inference latency while preserving a higher abstraction level than fully custom accelerators.

Although the DPU is configurable for different performance and resource constraints, its internal implementation is not normally exposed at RTL level. It is provided as a preconfigured component within the Vitis AI flow, so users mainly interact with model preparation, quantization, compilation, and deployment. This simplifies integration, but limits direct changes to the hardware microarchitecture.

B. Vitis AI Toolchain

Vitis AI is the AMD toolchain for deploying neural networks on DPU accelerators [14]. The flow starts from a model trained with common frameworks, such as PyTorch or TensorFlow, and transforms it into an executable representation for the target FPGA platform. Its main components are the quantizer and the compiler.

The quantizer converts floating-point models into reduced-precision formats, typically integer representations, suitable for efficient DPU execution. During this step, the model is calibrated with a representative dataset to estimate scales, numerical ranges, and quantization parameters for weights and activations. This phase affects both final accuracy and graph consistency.

The Vitis AI compiler analyzes the quantized model and generates the executable graph for the target platform. As part of this process, it produces compiled `.xmodel` artifacts that are compatible with execution on the DPU. DPU-compatible operations are identified and grouped into accelerable subgraphs, while unsupported operations are assigned to the host CPU. In the standard flow, the compiler processes the complete network and can apply global optimizations for efficient accelerator mapping. Overall, Vitis AI links the trained model to hardware execution through quantization, calibration, compilation, and the generation of deployment artifacts for DPU execution.

C. Xilinx Intermediate Representation

The Xilinx Intermediate Representation (XIR) is the graph-based intermediate representation used in Vitis AI to describe neural networks targeting hardware execution [17]. Each node represents an operation and stores metadata such as quantization parameters, tensor attributes, and compilation-related information. This structure preserves a consistent model representation when moving from high-level frameworks to the FPGA platform.

During compilation, the XIR graph is analyzed to identify subgraphs executable on the DPU and portions that must remain on the host CPU. XIR also supports optimizations such as operator fusion, for example merging convolution, normalization, and activation sequences into computation units better suited to hardware execution. Its APIs allow programmatic inspection and manipulation of the graph, including topology navigation, access to node attributes, and modification of the intermediate representation.

D. SLURM Workload Manager

SLURM is an open-source workload manager for cluster environments [18]. It allocates computational resources, launches processes on available nodes, and coordinates parallel or distributed applications. In a cluster, SLURM allows users to request nodes, CPUs, memory, accelerators, or other resources, and to execute programs through commands such as `srun`.

Operationally, SLURM separates resource management from the executed application. The system selects nodes, prepares the execution environment, and starts the processes, while communication between them remains managed by the application or runtime framework. This separation is useful when experiments must be launched, supervised, and repeated in a controlled way without embedding cluster-management logic directly into the application code.

IV. Methodology

The proposed framework automates the generation of DPU-optimized `.xmodel` fragments starting from a neural network, a calibration dataset, and a set of user-defined cut points. The objective is to provide a pipeline that, given the cut points, performs quantization, calibration, splitting, boundary correction, graph verification, and fragment compilation, while preserving as much as possible the optimizations obtained on the complete model.

A. Empirical Characterization of Fragment Compilation

Before defining the splitting framework, an empirical characterization of the Vitis AI compiler was carried out to observe the behavior of the toolchain when quantized XIR graphs are divided into independent fragments. For this purpose, about thirty heterogeneous small-to-medium micro-models were created. They were designed to reproduce common computational patterns in neural networks and to highlight the effects of splitting on sequential, branched, and residual structures.

The micro-models were organized into two main groups. The first group included linear convolutional patterns, such as `Conv-BN-ReLU` sequences, `Conv-ReLU` blocks, pooling operators, and combinations of consecutive layers. The second group included non-linear structures, such as parallel branches, residual connections, convergence points, and aggregation operators. This selection made it possible to evaluate the compiler behavior both on simple flows, where dependencies follow an almost sequential path, and on graphs where operator relations are distributed across multiple paths.

For each micro-model, the complete graph was compiled first and used as a reference to measure the mapping of operations onto the DPU. The same model was then split at intermediate or critical points of the graph. The cut points were selected to cross regions that are relevant for the compiler, for example between convolution and activation, between batch normalization and activation, between pooling and the following convolution, inside a parallel branch, or near aggregation nodes such as `add` and `concat`. The resulting fragments were then compiled separately and compared with the compiled version of the complete model.

This procedure made it possible to directly analyze which XIR graph transformations preserve efficient DPU mapping and which ones introduce optimization loss. In the case of the complete model, the Vitis AI pipeline has access to the whole quantized XIR graph. The compiler can therefore apply global optimizations, such as compatible-operator fusion, consistent management of numerical conversions, and mapping of supported regions onto the DPU. In this configuration, the ARM CPU is generally limited to secondary or unsupported operations, while the main computational workload is executed on the accelerator.

During the characterization, `fix` nodes emerged as a key element for the correct interpretation of fragments. These nodes do not correspond to layers of the original model, such as convolutions or activations, but are XIR operators introduced during quantization. They mark the transition of tensors to the fixed-point domain used by the DPU and store numerical metadata, including the `fix_point` parameter required to correctly interpret quantized values. Therefore, the presence of a `fix` node indicates that a tensor belongs to the quantized flow compatible with the accelerator.

When the graph is split, some tensors that were internal in the complete model become inputs or outputs of the fragments. During this step, part of the numerical and structural information associated with these tensors may remain outside the extracted sub-network. The metadata expressed by `fix` nodes, or implicitly encoded in producer–consumer relations, may be located in the previous or following fragment. As a consequence, a region that is fully compatible with the DPU in the complete model may be interpreted as incomplete when compiled in isolation.

TABLE I
DPU UTILIZATION FOR REPRESENTATIVE COMPILER PATTERNS.

Pattern	Cut	Complete (%)	Split (%)
Conv-BN-ReLU	BN/ReLU	71.4	50.0
Conv-ReLU	Conv/ReLU	71.4	50.0
Pooling	Pool/Conv	75.0	73.3
Branch split	In branch	85.7	63.6
Branch join	Add/Concat	88.2	68.0

This behavior was evident in cuts applied inside sequential patterns recognized by the compiler. In micro-models based on `Conv-BN-ReLU` or `Conv-ReLU` blocks, splitting between tightly connected operators reduced the compiler's ability to reconstruct the same fusions obtained on the complete graph. In these cases, the fragment boundary interrupts a relation that, in the full network, contributes to the generation of a more compact DPU region. Separate compilation may therefore introduce additional conversions or assign to the CPU operations that, in the complete model, were included in the accelerated subgraph.

In micro-models with branches, residual connections, and convergence points, the optimization loss was instead associated with the removal of lateral dependencies. In these graphs, the information required to correctly interpret a fragment is not always located along the main path delimited by the cut points. Operators such as `add`, `mul`, `scale`, or `concat` may depend on tensors produced along parallel paths. If these nodes or their predecessors are excluded, the compiler receives a structurally incomplete fragment and may not recognize some accelerable regions.

As shown in Table I, fragment compilation produces, in several cases, a reduction in DPU utilization compared with the compilation of the complete model. In the `Conv-BN-ReLU` and `Conv-ReLU` patterns, cutting at an intermediate point of the block reduces the percentage of operations mapped onto the accelerator. In branched cases, the reduction is mainly related to the loss of structural dependencies required to correctly reconstruct the original graph. Therefore, the cut point does not act only as a topological separation of the graph, but also changes the context used by the compiler to apply its optimizations.

The analysis of the results shows that the loss of acceleration does not necessarily depend on the presence of operators unsupported by the DPU. In several micro-models, the same operations are accelerated when the complete graph is compiled, but are partially assigned to the ARM Cortex-A53 when the model is split and compiled as independent fragments. Since the ARM processor is not optimized for highly parallel quantized workloads, this fallback can introduce a significant latency bottleneck.

This behavior is caused by the different level of information available during compilation. With the complete graph, the

compiler has access to the global relations between operators, the quantization nodes, and the dependencies between parallel paths. With independent fragments, some of this information may be located beyond the fragment boundaries. Under these conditions, the toolchain adopts a conservative strategy and prioritizes compilation correctness over maximum acceleration.

The characterization performed on the micro-models guided the design choices of the proposed framework. The results highlighted the need to perform splitting after global quantization, to explicitly reconstruct the numerical context at fragment boundaries, and to complete each subgraph with the required dependencies before compilation. The framework was therefore developed to generate independent fragments that preserve the structural and quantization information available in the complete model.

B. Post-Quantization XIR Fragmentation

After analyzing the behavior of the Vitis AI compiler when the graph is partitioned, a splitting framework was developed to reduce the loss of optimizations during independent fragment compilation. The framework takes as input the original FP32 model, a calibration dataset representative of the target application, and a set of user-defined cut points. Starting from these inputs, it generates independent XIR fragments intended for DPU execution, while preserving the quantization context and structural information available during complete-graph compilation.

As shown in Fig. 1, the proposed framework follows the standard Vitis AI flow and extends it with an intermediate phase dedicated to controlled XIR graph splitting. The FP32 model is first quantized and calibrated using the calibration dataset. This step produces a quantized XIR graph in which the numerical information required for DPU execution is derived from the complete model. Only after this global quantization phase does the framework apply the selected cut points, extract the corresponding fragments, reconstruct their boundaries, and prepare them for the following verification and generation stages.

Performing the split after quantization is a central design choice. If the network were partitioned before calibration, each fragment would be quantized separately, losing the common numerical reference of the original model. By quantizing the whole network first, the framework can recover from the complete XIR graph the quantization parameters needed to build fragments that are consistent with the fixed-point representation used by the DPU.

Starting from the quantized XIR graph, the framework analyzes the producer–consumer relations between operators. For each portion delimited by the selected cut points, the computational nodes belonging to the corresponding execution path are identified. These nodes form the initial core of the fragment. The framework then adds the auxiliary elements required for the correct interpretation of the fragment, such as constants, parameters, and quantization-related operators.

The boundary reconstruction phase restores the quantization context that may be lost when internal tensors become

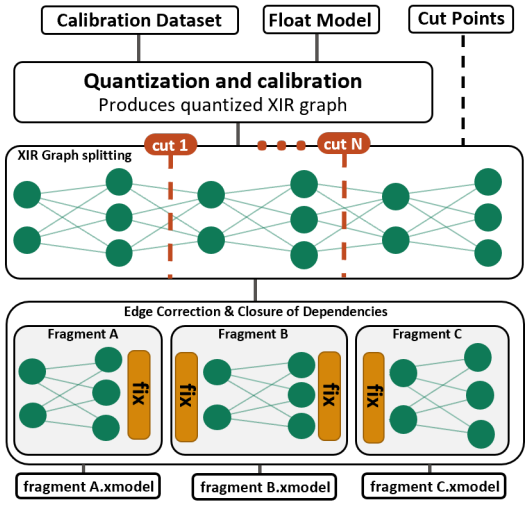


Fig. 1. Flow of the proposed framework for quantization, splitting, correction, and compilation of the fragments.

fragment inputs or outputs. Instead of treating these tensors as generic external values, the framework uses the quantization metadata available in the complete XIR graph to reconstruct the corresponding boundary representation. In practice, synthetic `fix` nodes are inserted at the input and output boundaries of each fragment, with parameters recovered from the original graph. For output boundaries, the framework traces the original computational path backward from the fragment output and uses the closest compatible `fix_point` value found in the complete graph. This allows each fragment to preserve a numerical representation consistent with its original position in the network.

After boundary reconstruction, the framework produces independent XIR fragments whose inputs and outputs are consistent with the quantization context of the original graph. Each fragment contains the selected computational core, the auxiliary nodes required for its correct interpretation, and the synthetic `fix` nodes inserted at the boundaries.

C. Fragment Completion and Structural Validation

After splitting and boundary reconstruction, the framework performs a structural verification of the generated fragments. This phase checks that each extracted graph is complete and can be serialized as an independent XIR model. In particular, it verifies that all operations required by the fragment are present in the generated graph and that no unresolved references remain toward portions excluded by the cut.

When a portion of the graph is extracted, it is not sufficient to copy only the operators located between two cut points. Each operation may depend on other nodes required for its execution, such as weights, biases, constants, quantization nodes, or results produced by previous operations. If one of these elements is not included in the fragment, the resulting graph may be incomplete or not correctly interpretable by the toolchain.

To avoid this issue, the framework performs dependency completion starting from the main set of operators assigned to the fragment. For each collected node, the incoming edges

are inspected in the original graph to identify any predecessors required to produce its inputs. If a required predecessor has not yet been copied, it is added to the fragment. The procedure is repeated until a complete iteration does not identify any new nodes to add to the fragment. Since the XIR computational graph is acyclic, this process terminates once all required dependencies have been included.

This phase is particularly useful in networks with residual connections, parallel branches, or local convergence points, where some dependencies may not lie on the main path of the fragment, while still being necessary for correct graph reconstruction.

Once the dependencies have been completed, the collected operations are ordered consistently with the graph dataflow. The framework therefore uses Kahn's algorithm to perform a topological ordering of the operators [19]. For this purpose, the in-degree of each node in the completed fragment is computed, and the ordering is initialized with the nodes that have no unresolved predecessors. These nodes are progressively removed from the working set, while the in-degree of their successors is updated. The resulting order defines the sequence in which operations are recreated in the new XIR graph, ensuring that each node is inserted only after the nodes it depends on.

This step ensures that the generated XIR graph is consistent, serializable, and correctly interpretable by the Vitis AI toolchain. Since the procedure is performed during offline fragment generation, it does not affect inference latency.

Once structural verification is complete, each fragment is saved as an independent XIR graph. The result of this stage is a collection of self-contained XIR models, each corresponding to a portion of the original network and ready for compilation through the Vitis AI toolchain. At the end of the process, the framework also generates the compiled models for DPU execution and creates a configuration file that describes the relation between the sub-models, specifying how the produced fragments must be connected and interpreted with respect to the structure of the original network.

V. Experimental Analysis

The results presented in this section validate the proposed framework from the perspective of DPU mapping. The analysis compares the behavior of the compiler with and without the corrections introduced by the framework, evaluating their impact on the distribution of operations between DPU and CPU and on the resulting execution latency. The evaluation considers two model families, a CNN with parallel branches and a ConvViT architecture, in order to assess the framework both on mainly convolutional partitions and on graphs with residual connections and more complex dependencies. The compatibility of the generated fragments with standard execution mechanisms, such as TCP and SLURM, is also evaluated.

A. Experimental Setup

The experiments were conducted on a cluster of three AMD Xilinx Kria KV260 boards, each equipped with a DPU DPUCZDX8G B4096 running at 300 MHz and a quad-core ARM Cortex-A53 processor at 1.33 GHz with 4 GB of DDR4 RAM. The boards are connected through a Gigabit Ethernet local network. The system software is Ubuntu 22.04 LTS (ARM64), with Vitis AI 3.5.0, VART and XIR in the same version, Python 3.10, and SLURM 21.08.5 for process orchestration. A separate x86 machine, external to the cluster, was used as a compilation server to run the Vitis AI toolchain inside a Docker container.

The first evaluated model family is ParallelExpertsNet, a pure CNN architecture with three parallel branches. All models accept input tensors with shape [1, 224, 224, 3] and produce an output with 10 classes. Four variants of increasing size were considered, whose architectural characteristics are reported in Table II.

TABLE II
Architectural characteristics of the evaluated CNN variants.

Variant	Input Size	Parameters (M)	Depth	Max Channels
Small	[1, 224, 224, 3]	∼1.3	12	128
Medium	[1, 224, 224, 3]	∼26	18	512
Large	[1, 224, 224, 3]	∼133	20	1024
Huge	[1, 224, 224, 3]	∼364.4	22	1536

The second evaluated family is based on ConvViT, with convolutional blocks and vision-transformer-like components. This architecture introduces a more complex graph than the pure CNN, including residual connections and aggregation operations that make the correct handling of internal fragment dependencies more relevant. The characteristics of the three considered variants are reported in Table III.

TABLE III
Architectural characteristics of the evaluated ConvViT variants.

Variant	Input Size	Parameters (M)	Embed Dim
Small	[1, 224, 224, 3]	∼2.3	192
Medium	[1, 224, 224, 3]	∼20	384
Large	[1, 224, 224, 3]	∼100	768

Each benchmark configuration was executed for 25 iterations. The first 5 iterations were used as warm-up and discarded from the final measurement, while the reported latencies correspond to the arithmetic mean of the following 20 executions. For each variant, fragments compiled without the framework, obtained through naive splitting, were compared against fragments generated with the proposed framework. The evaluation considers the percentage of operations mapped to the DPU and the overall execution latency. To verify the compatibility of the pipeline with standard orchestration tools, the same corrected fragments were executed both with manual launch and direct TCP communication, and through SLURM with FPGA resource allocation using the GRES mechanism.

B. Experimental Results

Before analyzing the effect of the framework on operation mapping, the compatibility of the pipeline with standard execution mechanisms was verified. This check was performed on the CNN models of the ParallelExpertsNet family, using the `.xmodel` fragments produced by the framework. The fragments were executed both with a manual TCP-based setup and through SLURM for process launch and orchestration. The goal of this comparison is to show that, once correctly compiled fragments are generated, their use does not require

dedicated hardware mechanisms or complex communication infrastructures.

TABLE IV
LATENCY COMPARISON BETWEEN MANUAL TCP AND SLURM-BASED ORCHESTRATION.

Variant	TCP (ms)	SLURM (ms)	Rel. (%)
Small	3.52	3.64	+3.4%
Medium	50.75	50.86	+0.2%
Large	122.65	122.54	-0.09%
Huge	322.89	323.02	+0.04%

The results in Table IV show that the two execution modes produce almost equivalent latencies. The relative differences are close to zero for Medium, Large, and Huge, while they are slightly more visible for Small, where the very low absolute latency makes any marginal variation more noticeable. This indicates that using an orchestrator such as SLURM is compatible with the proposed pipeline and allows a higher-level workflow without introducing significant overhead compared to manual TCP management.

The second analysis evaluates the effect of the framework on DPU operation mapping. The DPU utilization results are reported in Table V. For ParallelExpertsNet, naive splitting reduces DPU utilization to 50–56%, since some boundary operations are excluded from accelerator mapping. After applying the framework, utilization increases up to 90.9%, confirming that correcting the boundary quantization context restores the expected mapping.

TABLE V
DPU UTILIZATION BEFORE AND AFTER APPLYING THE FRAMEWORK.

Arch.	Variant	Without (%)	With (%)
CNN	Small	50.0	84.6
CNN	Medium	50.0	84.6
CNN	Large	56.0	84.6
CNN	Huge	56.0	90.9
ConvViT	Small	4.2	95.6
ConvViT	Medium	4.2	97.6
ConvViT	Large	4.2	86.1

The ConvViT models show a different issue. Without the framework, DPU utilization remains at 4.2% for all variants, indicating that naive splitting generates fragments that are not sufficiently complete from the compiler's point of view. In this case, the problem is not limited to `fix` nodes at the boundaries, but also involves the collection of internal dependencies introduced by residual connections and aggregation operations. With the framework, dependency closure reconstructs structurally consistent fragments, increasing DPU utilization up to 97.6% for the Medium variant.

The latency impact for ParallelExpertsNet is shown in Fig. 2. The logarithmic scale highlights how naive splitting makes execution dominated by residual CPU operations, producing latencies up to two orders of magnitude higher than the corrected version. The most critical case is the Huge variant, which goes from 80,349 ms without the framework to 322.89 ms after correction, corresponding to a ×249 slowdown. The degradation is also significant for intermediate models, with slowdowns of ×162 for Medium and ×150 for Large.

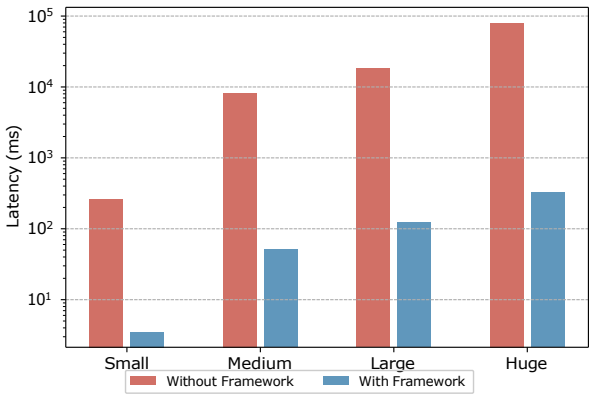


Fig. 2. ParallelExpertsNet inference latency before and after applying the framework, shown on a logarithmic scale.

The latency impact for ConvViT models is reported in Fig. 3. In this case as well, naive splitting produces a strong performance degradation, but the main cause is related to missing structural dependency collection in blocks with residual connections. The Small variant goes from 451 ms without the framework to 4.79 ms after correction, with a slowdown of about ×94, while the Medium variant goes from 3,521 ms to 39.37 ms. The most critical case is the Large variant, which goes from 16,997 ms to 31.36 ms, with a slowdown up to about ×542.

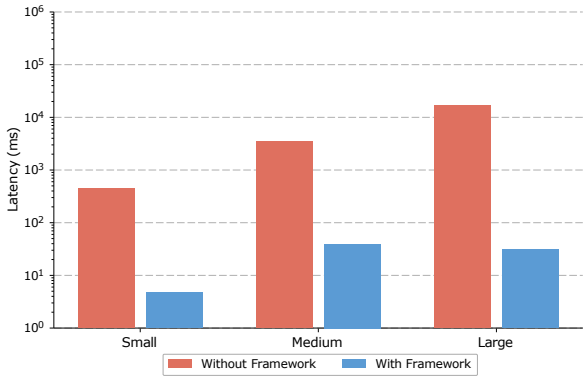


Fig. 3. ConvViT inference latency before and after applying the framework, shown on a logarithmic scale.

Overall, these results confirm that naive splitting can significantly degrade fragment compilation, but for different reasons depending on the model structure. In CNN models, the main issue is the loss of boundary quantization context, while in ConvViT models the dominant issue is the missing inclusion of dependencies generated by residual connections and aggregation operations. The proposed framework addresses both aspects, restoring effective DPU mapping and bringing latencies back to the expected range for hardware-accelerated execution.

The improvement in accelerator mapping introduces an offline preparation cost, reported in Table VI. This cost corresponds to the complete pipeline required to obtain `.xmodel` fragments suitable for deployment, and includes calibration,

quantization, splitting, and compilation. The evaluated configurations cover four different model sizes, ranging from approximately 1M to approximately 400M parameters.

TABLE VI
FRAMEWORK EXECUTION TIME IN SECONDS AS A FUNCTION OF MODEL SIZE.

Stage	~1M params	~25M params	~100M params	~400M params
Calibration	6.39	28.13	82.29	163.95
Quantization	0.41	2.29	11.70	32.51
Splitting	0.08	0.76	5.37	20.26
Compilation	0.95	6.26	36.01	247.69
Total	**7.83**	**37.44**	**135.37**	**464.41**

As shown in the table, the preparation time increases with model size. The complete pipeline requires 7.83 s for a small model with approximately 1M parameters and reaches 464.41 s for the largest model with approximately 400M parameters. The custom splitting stage remains limited compared with the other stages, requiring only 20.26 s even in the largest case, while calibration requires 163.95 s and compilation requires 247.69 s. This shows that the overhead introduced by the splitting step is limited compared with the overall cost of the Vitis AI pipeline.

VI. CONCLUSIONS AND FUTURE WORK

This work presented an XIR-level framework for splitting neural networks into independently compiled fragments while preserving effective DPU mapping. The empirical analysis of the Vitis AI compiler showed that naive splitting can remove quantization and structural information required for optimized compilation, leading to CPU fallback and significant latency degradation.

The proposed framework operates on the quantized XIR graph before compilation. Starting from a complete model, a calibration dataset, and user-defined cut points, it generates independent `.xmodel` fragments by reconstructing the quantization context through `fix` nodes and completing the structural dependencies required by each fragment. This allows the optimizations of the original model to be preserved without modifying the DPU or introducing additional hardware logic.

Experimental results on ParallelExpertsNet and ConvViT show that naive splitting can produce slowdowns up to ×249 on CNN-based models and up to about ×542 on ConvViT models. In both cases, the framework restores effective DPU mapping and brings latency back to the expected range for hardware-accelerated execution. The comparison between manual TCP execution and SLURM execution also confirms that the generated fragments can be deployed with standard tools while maintaining a high-level workflow.

Future work will focus on making the framework fully autonomous. A first direction is the automatic selection of cut points, so that the graph can be partitioned to maximize parallelism while preserving compiler optimizations. Another objective is to improve the distribution of fragments across the available computational nodes, avoiding unbalanced executions in which most of the workload is assigned to a single node. In this way, the framework could evolve from a user-guided splitting pipeline into an automatic partitioning and deployment tool for distributed DPU-based inference.

REFERENCES

[1] F. Buccellato, E. Vacca, S. Azimi, C. De Sio, and L. Sterpone, "From Detection to Intervention: An End-to-End System for Recognizing the "Signal for Help" Gesture in Real-Time," *Intelligent Systems with Applications*, vol. 26, Art. no. 200536, 2025, doi: 10.1016/j.iswa.2025.200536.

[2] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks," in *Advances in Neural Information Processing Systems*, vol. 25, 2012, pp. 1232–1240.

[3] W. Jiang, E. H.-M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, "Achieving Super-Linear Speedup across Multi-FPGA for Real-Time DNN Inference," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5s, pp. 67:1–67:23, Oct. 2019, doi: 10.1145/3358186.

[4] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[5] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[6] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks," in *Proc. 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey, CA, USA, Feb. 2015, pp. 161–170, doi: 10.1145/2684746.2689060.

[7] G. Cora, E. Vacca, C. De Sio, S. Azimi, and L. Sterpone, "Fast SEU Detection and Recovery in FPGA-Based AI Accelerators," *ACM Transactions on Reconfigurable Technology and Systems*, 2026, doi: 10.1145/3806052.

[8] G. Cora, D. Rizzieri, C. De Sio, S. Azimi, and L. Sterpone, "In-Hardware Fault-Tolerance Controller for Multi-FPGA Clustered Architectures," *IEEE Transactions on Computers*, 2026, doi: 10.1109/TC.2026.3709831.

[9] Y. Zhu, Z. He, W. Jiang, K. Zeng, J. Zhou, and G. Alonso, "Distributed Recommendation Inference on FPGA Clusters," in *Proc. 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 279–285.

[10] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 65–74.

[11] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From High-Level Deep Neural Models to FPGAs," in *Proc. 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 17:1–17:12.

[12] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A Toolflow for Mapping Diverse Convolutional Neural Networks on Embedded FPGAs," *arXiv preprint arXiv:1711.08740*, 2017.

[13] AMD, "Deep Learning Processing Unit," in *Vitis AI User Guide (UG1414)*, Version 3.5, Sep. 28, 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug1414-vitis-ai

[14] AMD, "Vitis AI User Guide (UG1414)," Version 2.5, 2022. [Online]. Available: https://docs.amd.com/r/2.5-English/ug1414-vitis-ai

[15] F. Buccellato, C. De Sio, S. Azimi, and L. Sterpone, "On-Hardware Resilience Analysis of DPU-Accelerated CNNs on FPGA-Based Systems," in *Proc. 28th Euromicro Conference on Digital System Design (DSD)*, Salerno, Italy, 2025, pp. 34–41, doi: 10.1109/DSD67783.2025.00017.

[16] F. Buccellato, C. De Sio, S. Azimi, and L. Sterpone, "Hardware-Aware Runtime Detection of Soft-Error Anomalies in DPU-Accelerated Neural Networks," in *Proc. IEEE International On-Line Testing Symposium (IOLTS)*, 2026.

[17] AMD, "Xilinx Intermediate Representation," in *Vitis AI User Guide (UG1414)*, Version 3.5, Sep. 28, 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug1414-vitis-ai/XIR

[18] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer, 2003, pp. 44–60, doi: 10.1007/10968987_3.

[19] A. B. Kahn, "Topological Sorting of Large Networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, 1962, doi: 10.1145/368996.369025.

An Exploratory Study on Code Smells Detection and Refactoring using LLMs

Giorgia Paisi
University of Milano-Bicocca
Milano, Italy
g.paisi@campus.unimib.it

Francesca Arcelli Fontana
University of Milano-Bicocca
Milano, Italy
francesca.arcelli@unimib.it

Bartosz Walter
Poznań University. of Technology
Poznań, Poland
bartosz.walter@cs.put.poznan.pl

***Abstract*—With the observed progress in machine learning (ML), and particularly the introduction of Large Language Models (LLMs), several activities related to code maintenance could be automated. That includes not only detection and evaluation of design flaws, but also code transformation and refactoring. However, the general-purpose LLMs, while being commonly used and popular, have not been specifically trained for code analysis, and may not be suitable for conducting software maintenance tasks due to biases, and inherent shortcomings of the models. In this paper, we explore if the widely available LLMs could aid the detection and the refactoring of code smells. We focus on four common smells (God Class, Long Method, Feature Envy, and Refused Bequest) and consider five prompts of diverse complexity, asking the model for detecting and removing the identified code smells. Results suggest that general-purpose LLMs cannot be reliably used for that. They can effectively detect or remove code smells only in simple cases, and frequently produce invalid code. However, their performance depends on various factors, e.g., the model, the specific code smell or the prompt objective and composition.**

***Index Terms*—code smells refactoring, code smells detection, LLM**

I. Introduction

Code smells are surface indicators of possible code or design problems in software systems [1]. Compared to code metrics, they provide a more abstract, but also more holistic view on maintainability: while metrics deliver narrowly-focused, objective values that require interpretation [2], code smells deliver a comprehensive description of the issue and its visible high-level manifestations, making their interpretations much more direct [3]. That way, code smells fill the gap between metric values and comprehensive code analysis [4].

Various approaches to detecting code smells have been proposed, starting from programmers' intuition and experience [1], through quantifiable expressions based on selected metrics [5], [6], analysis of syntactic or dependency graphs [7], tracing the history of code change [8], machine learning-based techniques [9], up to exploiting the knowledge of common relationships among smells [10]. In response, various smell detecting tools have been proposed; however, their convergence remains relatively low [11]. The advent of Large Language Models (LLMs) paved the way for novel techniques in code processing and analysis [12]. Despite several drawbacks regarding contextual awareness and reliability [13], LLMs offer an interesting alternative to the classical approaches to transforming various software artifacts [14], [15].

The goal of this study is to explore whether general-purpose LLMs could be reliably used for code smell analysis. Specifically, we assess their capabilities in both detecting and refactoring four distinct code smells: God Class, Long Method, Feature Envy, and Refused Bequest. By evaluating these dual capabilities, this study reveals whether general-purpose models are effective for practical, end-to-end applications in software maintenance.

II. Related Work

Despite the broad availability of tools, code smells are often left unresolved: according to Shetty et al. [16], over 15% of the detected smells remain unchanged.

LLMs in Code Smell Detection: LLMs' capabilities in code smell detection have recently been assessed by Silva et al. [12]. They found that ChatGPT's performance varies depending on the prompt composition: while detailed prompts contribute to the correct detection, the overall accuracy remains low. In a similar study, Sadik and Govind [17] compared GPT-4.0 and DeepSeek-V3, to find that while GPT-4.0 consistently generated fewer false positives, both models exhibited relatively low recall.

General-Purpose LLM-Driven Refactoring: Recent works studied the use of general-purpose LLMs for conductinf the maintenance tasks. Midolo and Tramontana [18] evaluated ChatGPT's effectiveness in refactoring Java *for* loops into stream pipelines. While it can successfully refactor several cases, its performance is inconsistent, e.g., it struggles with complex loops that require deep contextual understanding. Liu et al. [19] assessed ChatGPT's and Gemini's ability to identify refactoring opportunities and suggest refactoring solutions. ChatGPT produced refactoring solutions that were comparable to or better than the original developers' code in 63.6% of cases, while Gemini achieved this in 56.2% of cases. Karabiyik and Abdulhamid [20] adopted a different approach, introducing RefactorGPT, a ChatGPT-augmented multi-agent

Manuscript received April 29, 2026; revised July 22, 2026; accepted July 16, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.145

framework that coordinates four specialised agents, each responsible for a specific task in the refactoring process: Analyzer, Refactor, Refine, and Fixer. This layered design allows a controlled execution and stepwise debugging of the refactoring process.

Fine-Tuned and Domain-Specific LLM-Driven Refactoring: Alazba et al. designed and implemented SmellyBot, a detection tool [21] based on a transformer model pre-trained on large-scale datasets. SmellyBot reported high precision for Data Class, God Class, Feature Envy, and Long Method smells. To support the fine-tuning of LLM and traditional machine learning techniques, Alomari et al. [22] developed a high-quality dataset of code smells with multi-label classification. Wu et al. [15] proposed iSMELL, a MoE (Mixture of Expert) architecture for code smell detection that integrates various specialised toolsets and leverages LLMs to refactor identified code smells. Their evaluation demonstrated that iSMELL outperforms other baselines in detection accuracy and satisfactorily refactors code smells while improving overall code quality attributes. Cordeiro et al. [14] compared refactorings made by humans with those performed by a StarCoder2 LLM-based tool. They reported that the LLM reduced the number of code smells by 44.36%, i.e., 20.1% more than human developers. On the other hand, the LLM-generated code achieved only a 57.15% test pass rate. Moreover, Cordeiro et al. [13] highlighted the limitations of LLM-generated refactorings. They noted that integrating LLMs into IDEs remains difficult due to several factors: limited contextual awareness, a lack of deep understanding regarding the refactoring process, the frequent unavailability of test cases to verify the logic of generated code, and persisting hallucinations.

Although prior work identifies LLMs as promising instruments for code smell detection and refactoring, the results are still fragmented. Many existing studies assess only a narrow subset of code smells and rely on relatively simple prompting approaches. There is no comprehensive investigation of how specific prompt engineering techniques systematically interact with different model architectures across a broad spectrum of code-smell categories; in particular, it is unclear how diverse prompting strategies perform when combined with varying architectures on different types of code smells. This study addresses that gap by conducting a multidimensional analysis that yields a fine-grained understanding of how model-specific capabilities can be exploited to tackle diverse refactoring challenges.

III. Study design

We aim to answer two research questions, focused on the detection and refactoring of code smells.

RQ1: Are LLMs capable of *detecting* code smells?

RQ2: Are LLMs capable of *refactoring* code smells?

The research questions aim to evaluate the effectiveness of LLMs in the code smell-guided maintenance activities. We examine their ability to distinguish between smelly and non-smelly code instances, specifically investigating whether models tend to perform unnecessary refactorings on smell-free code. Furthermore, we analyze how their performance is affected by the specific type of code smell, the prompt engineering technique employed, and the underlying model architecture.

A. Code smells and datasets

We focus on four code smells: God Class (GC), Long Method (LM), Feature Envy (FE), and Refused Bequest (RB), introduced by Fowler [1]. These smells have been extensively studied and documented in the literature [4], [11], [23], and various approaches to the detection and refactoring them have been proposed [24]–[27]. The subject smells capture various violations of coding principles, e.g., concerning cohesion, coupling, and inheritance, at the class and method level. Additionally, they are reliably detected by several static analysis tools, which makes them practical choices for empirical investigation.

To identify a suitable dataset of projects, we followed recommendations by Zakeri et al. [28], and then narrowed down the selection to projects with validated code smell instances of GC, LM, FE, and RB, located in publicly available Java projects with available test suites. As a result, we chose a dataset provided by Cruz et al. [29][1],with 252k non-smelly instances, and 14,918 smelly ones, detected with four code smell detectors: Organic[2] (for GC, FE, and RB smells), JSpIRIT [5][3] (for GC, FE, and RB), JDeodorant[4] (for GC, LM, and FE), and Designite[5] (only for LM). While the original dataset exhibits a significant class imbalance (252,000 non-smelly vs. 14,918 smelly instances), we implemented a controlled sampling strategy to mitigate the potential bias. Specifically, we performed a stratified analysis of approximately 1,300 instances per code smell (GC, LM, FE, and RB), maintaining a balanced 1:1 ratio between the smelly and non-smelly cases to avoid a distorting precision and recall by the majority class distribution.

B. Models, prompts used and evaluation methods

To ensure diversity of data, we reviewed various LLMs, focusing on the popular, general-purpose ones which offered sufficient context sizes at the time of designing the study (March 2025). Based on that, we decided to use four models: *Deepseek-chat*, *Gemini 2.0 Lite*, *Gemini 2.0 Flash*, and *Gemini 1.5 Pro*. Various prompting patterns for LLMs are in common use. In this study, we use five *refactoring-oriented* prompts (R1-R5) and one *detection-oriented* prompt (D1). This selection aims to evaluate the trade-off between the minimalist guidance and structured heuristics. We aim to analyze each model's response and determine whether there is a relationship between the use of a specific pattern and the effects of refactoring.

The prompts are described below.

[1]https://github.com/DVSCross/BadSmellsDetectionStudy
[2]https://github.com/diegocedrim/organic
[3]https://github.com/hcvazquez/JSpIRIT
[4]https://users.encs.concordia.ca/~nikolaos/jdeodorant/
[5]https://github.com/tushartushar/DesigniteJava

Detection-oriented prompt:

- **D1**: A *zero-shot* detection prompt describing the code smell.

Refactoring-oriented prompts:

- **R1**: A *Zero-shot* prompt. It describes the specific code smell and its usual refactorings.
- **R2**: A *One-shot* prompt. It describes the specific code smell, its usual refactorings, and provides a single example of the refactoring strategies.
- **R3**: A *chain-of-thought (COT)* prompt. It includes a step-by-step process to guide the LLM processing.
- **R4**: A *Rule-based* prompt. It contains a series of rules [30] to apply to each possible refactoring.
- **R5**: A *Context* prompt. It only contains a definition of the term "refactoring".

While the D1 prompt focuses solely on the LLMs' detection performance, the refactoring prompts R1-R5 serve the dual purpose of detecting and refactoring the smells. They explicitly instruct the models to apply a refactoring only if the smell is present in the given instance.The verbatim prompts are included in the replication package.

C. Workflow

For each instance in the dataset:

1) Append the instance to six prompts, and send each pair [Prompt, Instance] to all LLMs;
2) Verify if the LLM's response is correctly labeled as *smelly* or *non-smelly*.
3) Only for the refactoring-oriented prompts: 1) Test the LLM-refactored code to verify if it still compiles, and if the tests still pass. 2) Apply the code smell detectors to verify if the smell has been correctly removed.

Evaluation methods: Based on the results, we assigned each instance to one of the following categories:

Resolved: The smell has been removed, and none of the tools detects it anymore.

Enhanced: The smell has been partially removed and is still detected by one or more tools.

Nothing changed: None of the tools detected any difference between the original and the refactored code.

Worsened: The smell is reported by more tools than before.

To classify the instances, we relied on the same smell detectors that were employed to construct the dataset. To reduce the likelihood of circular validation bias, we applied four distinct code smell detectors (as outlined in III-A) and marked an instance as "resolved" only if none of the detectors reported the presence of the smell.

IV. Results

A. Detection performance (RQ1)

To analyse LLMs' detection capabilities, we used one of the detection prompts with both smelly and non-smelly instances. These results were compared against those from the refactoring prompts, which instructed the models to verify the presence of a code smell before attempting a refactoring, to verify whether LLMs prevent unnecessary modification of non-smelly instances. *DeepSeek* failed to identify any smells and, for each instance it responded *"The smell is not present"*. For each model, when using a D1 prompt, as follows from Table I, both *Precision* and *Recall* remained $\leq$0.5.

Model	Precision	Recall	F1
Gemini Lite	0.45	0.17	0.25
Gemini Flash	0.31	0.17	0.22
Gemini Pro	0.50	0.46	0.48

TABLE I
DETECTION PERFORMANCE OF DIFFERENT MODELS (FOR THE D1 PROMPT).

	Detection Prompt			Refactoring Prompt		
Smell	Precision	Recall	F1	Precision	Recall	F1
GC	1.00	0.57	0.72	0.67	0.93	0.78
LM	0.17	0.83	0.28	0.55	0.94	0.69
FE	0.28	0.12	0.16	0.68	0.71	0.69
RB	0.76	0.21	0.33	0.71	0.89	0.79

TABLE II
COMPARISON OF DETECTION PERFORMANCE BETWEEN THE D1 AND R1-R5 PROMPTS

The left side of Table II shows the results for the D1 prompt, organized by code smell. In terms of *Precision*, the detection of *GC* and *RB* performs best, achieving scores of 1 and 0.76, respectively. In contrast, *LM* and *FE* are below 0.30. Regarding *Recall*, it is different: *LM* achieves the best result with 0.83, followed by *GC* at 0.57. Both *FE* and *RB* barely reach 0.20, demonstrating limited *Recall* capabilities. The *F1 score* highlights how *GC* achieves the best overall performance (0.72).

R1-R5 prompts, even though the detection step was only linked to a simple conditional statement, deliver better results. In this case, as shown in the right side of Table II, *F1 scores* for all code smells are between 0.69 and 0.79. *RB* has the best *Precision* (0.71), while *GC* and *FE* show similar values (0.67 and 0.68 respectively). *LM* ranks lowest with 0.55. In terms of *Recall*, *LM* and *GC* reach 0.94 and 0.93, respectively. *RB* follows closely with 0.89, while *FE* is the lowest, at 0.71. These results indicate that the use of a refactoring-oriented prompt improves the LLMs' detection performance and yields more consistent results compared to the detection-specific approach.

Regarding RQ1, we noticed that LLMs struggle in consistently detecting code smells, with negligible differences when using a detection-specific prompt. Better performance is observed when the prompt focuses on a different task (e.g., refactoring) and detection occurs as a side effect. When using a detection-oriented prompt, the ability of LLMs to detect code smells is strongly dependent on the specific type of smell, with *GC* achieving the highest *F1 score*. In contrast, when performing detection with a refactoring-oriented prompt, all code smells have improved *F1 scores*, and the LLMs perform consistently better across different smells. These results are interesting, as refactoring-oriented prompts only contained a

simple conditional statement concerning the smell detection. It is important to note that the study primarily focuses on refactoring of code smells, and only one detection-specific prompt was used. Consequently, the ability of LLMs to discern when refactoring is necessary (and when to refrain) emerges as a contextual strength rather than a simple pattern-matching capability, justifying our multidimensional focus on the interplay between task framing and model architecture.

B. Refactoring performance (RQ2)

Even though the LLMs were asked to refactor both smelly and non-smelly instances, this section focuses on the refactoring results for smelly instances. As reported in Sec. IV-A, R1-R5 prompts yielded considerable detection results, indicating that they rarely attempt to refactor non-smelly code. To analyze the refactoring performance and address **RQ2**, it is important to consider the *build results*: the percentage of code generated by the LLMs that compiles successfully and passes all unit tests in their project's test suite. To analyse DeepSeek's capabilities alongside those of the other models, the explicit detection request was removed from its prompts, while keeping the remainder unchanged. Since any difference between DeepSeek's and Gemini's prompts might affect the comparability, the prompt was only changed only by removing a short statement sentence. This alteration, however, poses a threat to validity of results. Within the Gemini family, the performance differences between models are consistent, showing a progressive improvement that aligns with each model's capabilities. The build success rate is highly dependent on both the code smell and the model, as shown in Table IV: percentages vary for the same model, when refactoring different code smells. Contrarily, the relationship between the chosen prompt and the build success rate appears less evident (Table V): percentages remain similar when the refactoring is executed by the same model. None of the models has an overall build success rate above 50%, and significant debugging skills are required to identify defects in the generated code. To further evaluate the refactoring performance, only the code successfully built was analyzed, and no manual debugging was performed. Each refactoring was categorized as described in the Sec. III-C.

Models	Successful Builds
Gemini Lite	31.11%
Gemini Flash	32.89%
Gemini Pro	42.13%
DeepSeek	17.91%

TABLE III
BUILD SUCCESS RATES IN DIFFERENT MODELS (REFACTORING-ORIENTED PROMPT)

Qualitative analysis: A manual inspection of the LLM-generated code for the failed builds revealed several recurring issues. Models frequently attempted to access private methods from external classes, failing to account for Java's access modifiers. In several instances, the models changed the visibility of original classes from public to private, resulting in compilation failures. Another frequent source of defects was the LLMs' tendency to hallucinate variables and method return types; this led to attempts of casting objects to incompatible types and pass incorrect argument types to constructors and methods. Furthermore, during refactoring, models often renamed methods intended to override a supertype, thereby breaking the inheritance chain and disrupting polymorphic behaviour. Finally, LLMs frequently hallucinated non-existent packages, methods, and variables. These patterns suggest that while the models understand local scope, they struggle to grasp the wider context of the project and often produce low-quality code.

Code Smells	GC	LM	FE	RB
Gemini Lite	25.21%	29.23%	56.31%	20.27%
Gemini Flash	17.36%	27.47%	63.47%	22.91%
Gemini Pro	16.31%	68.73%	42.57%	23.67%
DeepSeek	0.95%	15.46%	46.42%	8.01%

TABLE IV
BUILD SUCCESS RATES IN DIFFERENT MODELS FOR DIFFERENT CODE SMELLS (REFACTORING-ORIENTED PROMPT)

Prompts	0-shot	1-shot	COT	Rule-based	Context
Gemini Lite	25.44%	26.43%	27.08%	31.16%	41.24%
Gemini Flash	31.93%	35.11%	22.53%	30.08%	42.20 %
Gemini Pro	41.45%	38.85%	39.87%	35.97%	50.06%
DeepSeek	16.32%	17.64%	16.23%	16.81%	22.29%

TABLE V
BUILD SUCCESS RATES IN DIFFERENT MODELS FOR DIFFERENT PROMPTS (REFACTORING-ORIENTED PROMPT)

Is there a difference in the refactoring capabilities based on the smell?

Table VI reveals a correlation of the refactoring capability with the specific code smell ($p < 0.001$, Pearson's Chi-square test). When refactoring *LM*, LLMs tend to perform better: in 18.75% of the cases, the code quality is *Enhanced*, and in 44.02% of the cases, the smell is entirely *Resolved*. Only 6.92% of the time the code quality is *Worsened*. This is different for *GC* and *FE*: both show a rate of *Nothing changed* exceeding 50%, indicating that quality has neither increased nor worsened. Even so, in both cases, the percentage of *Solved* is comparable to the percentage of *Worsened* cases (16.04% vs. 1.86% for *GC*, and 13.71% vs. 19.45% for *FE*). *RB* shows the poorest performance among all code smells. Although the successful build rate is comparable to the others, the refactoring produces little to no improvement in the code.

Code smells	GC	LM	FE	RB
Resolved	16.04%	44.02%	13.71%	1.18%
Enhanced	11.19%	18.75%	6.99%	0%
Not. changed	70.89%	30.29%	59.83%	98.09%
Worsened	1.86%	6.92%	19.45%	0.71%

TABLE VI
PERFORMANCE OF REFACTORING SPECIFIC SMELLS WITH ALL PROMPTS

Is there a difference in the refactoring capabilities based on the prompt?

Prompts	0-shot	1-shot	COT	Rule*	Context
Resolved	20.64%	20.14%	25.65%	16.15%	21.06%
Enhanced	10.60%	10.53%	6.79%	9.98%	10.10%
Not. changed	52.65%	53.97%	59.64%	60.07%	60.17%
Worsened	16.09%	15.34%	7.89%	13.79%	8.65%

TABLE VII
PERFORMANCE OF REFACTORING ALL SMELLS WITH SPECIFIC PROMPTS. *Rule: Rule-based*

Table VII reports the refactoring performance for different prompts. It is difficult to identify a clear relationship between the prompt type and refactoring performance. All prompts show the *Nothing Changed* rate between 50% and 60%, with the percentage of *Solved* exceeding that of *Worsened* quality. The only significant differences are related to the COT, which yields the best results ($p < 0.001$, Pearson's Chi-squared test) with the highest *Solved* and lowest *Worsened* rate, and Context, which has the highest *Nothing Changed* rate of 60.17%. The difference appears to be linked to the level of detail in the prompts: the prompts that exhibit the best performance are also those that best explain the smell and its refactoring.

Is there a difference in the refactoring capabilities based on the LLM?

Models	G. Lite	G. Flash	G. Pro	DeepSeek
Resolved	12.26%	17.58%	12.63%	55.37%
Enhanced	7.51%	7.72%	17.89%	7.98%
Not. changed	71.60%	59.23%	53.89%	32.78%
Worsened	8.61%	15.45%	15.57%	3.85%

TABLE VIII
PERFORMANCE OF REFACTORING ALL SMELLS WITH ALL PROMPTS USING SPECIFIC MODELS

Table VIII reports the refactoring performance by model. The data shows that *DeepSeek* outperforms all Gemini models ($p < 0.001$, Pearson's Chi-squared test), which exhibit comparable performance, exceeding their *Resolved* rates by a factor of three. However, these results lack comprehensiveness: *DeepSeek* has a low *Build Success* rate, with most successful compilations occurring for *LM* and *FE*. It is therefore difficult to estimate *DeepSeek*'s true refactorings' efficacy, as these smells appear to be the easiest to refactor and the data are unevenly distributed. The three Gemini models exhibit similar results overall: *Gemini Pro* produces better refactorings, but also a higher share of *worsened*-quality code. The differences among the Gemini models appear to be consistent with their cost and abilities.

A closer inspection of Table IX, reveals another pattern: each model's refactoring performance varies depending on the specific code smell. *DeepSeek* achieves a 96.05% success rate when refactoring *LM* and a 52.24% success rate for *FE*, making it the best-performing model for these two code smells. Conversely, *DeepSeek* shows the worst results for *GC* and *RB*, with a 0% success rate. These data, together with its 17.91% *Build Success* rate, make this model the least suitable for refactoring code smells. In contrast, all three Gemini models exhibit different performance across the various code smells. The *Solved* and *Enhanced* rates (cases when an improvement was detected) of **GC** are consistent with model ranking: *Gemini Lite* has the lowest value (3.63% and 11.81%), *Gemini Flash* falls in the middle (26.42% and 10%), and *Gemini Pro* achieves the highest (15.38% and 23.07%). Both *Nothing Changed* and *Worsened* rates follow similar patterns, decreasing as the model improves. Concerning **LM**, although *Gemini Lite* and *Gemini Flash* follow the trend described above, with 37.55% and 69.27% *Resolved* rates respectively, *Gemini Pro* shows significantly lower performance, with only 17.53% of *Resolved* instances. *Gemini Pro* also exhibits the highest *Worsened* rate. When refactoring *FE* all three models deliver similar results, with comparable *Resolved* and *Nothing Changed* rates. Finally, to analyze LLMs' capabilities to refactor **RB** the results reported in Table VI are fundamental: RB is the least-refactored code smell, with a 100% *Nothing changed* rate for both *Gemini Lite* and *Gemini Pro*, and a 96.33% rate for *Gemini Flash*. Although *Gemini Flash* exhibits better results, only 2.29% of the code smells are *Resolved*.

Smells	GC	LM	FE	RB	GC	LM	FE	RB
Models	**Gemini Lite**				**Gemini Flash**			
Resolved	3.63	37.55	6.23	0	26.42	69.27	5.43	2.29
Enhanced	11.81	19.24	3.23	0	10.0	12.84	8.48	0
Not. changed	82.72	34.27	77.36	100	61.42	17.87	57.55	96.33
Worsened	1.81	8.92	13.16	0	2.14	0	28.52	1.37
	Gemini Pro				**DeepSeek**			
Resolved	15.38	17.53	6.21	0	0	96.05	52.24	0
Enhanced	23.07	26.49	6.21	0	0	3.94	10.61	0
Not. changed	61.53	44.02	63.84	100	100	0	31.42	100
Worsened	0	11.94	23.72	0	0	0	5.71	0

TABLE IX
LLMS' RESULTS (IN PERCENT) OBTAINED REFACTORING SMELLS WITH ALL PROMPTS.

Qualitative analysis: Refactoring code with an LLM involves trade-offs among different types of code smells, as a manual analysis of results from all categories (Solved, Enhanced, Nothing Changed and Worsened) revealed. For instance, refactoring *LM* instances resolves *Complex Method* and *Long Statement* smells, but introduces *Magic Numbers* and *Long Parameter Lists* in return. Similarly, the refactoring of *GC* increases the number of *FE*, suggesting that LLMs struggle to maintain proper encapsulation when decomposing large classes. Moreover, *RB*'s refactoring occasionally resolves *FE* only to introduce *DC* smells.

With respect to **RQ2**, the refactoring capabilities of LLMs depend on the model and the specific code smell. Accuracy is affected by the chosen prompt, with more detailed prompts performing better. As described in Table III, these data must be interpreted in light of the *Build Success* rates: none of the models exceeds 50% in successful builds.

V. Discussion

Our results indicate that the subject models were unable to consistently detect code smells when the detection was their only task. This finding aligns with the works by Silva et al. [12] and Sadik and Govind [17], who reported that LLMs detection performance is heavily prompt-dependent. Similar to Sadik and Govind's observations of GPT-4.0, the Gemini models in our study generally achieved higher precision than recall (Table I), indicating a conservative tendency to overlook code smells. However, when using refactoring-oriented prompts, the result reversed, and recall increased significantly. Notably, the F1-score improved alongside precision, signaling that refactoring-oriented instructions help the model identify smells more accurately without increasing false positives.

We achieved a moderate *successful builds* rate between 17% and 42%. This is notable, as Cordeiro et al. [14] reported a first-pass performance of 28.4% using a StarCoder2, a model specifically trained for code generation. Our results demonstrate that general-purpose LLMs could achieve refactoring capabilities comparable to those of models trained for code generation. As described in Sec. IV-B, model performance depends on the specific code smell. The best results were obtained for *LM*, while *RB* was the most challenging. Cordeiro et al. [14] observed similar results, reporting better performance for syntactic refactoring tasks compared to more complex refactoring requiring broader contextual understanding. Furthermore, this mirrors Midolo and Tramontana's [18] observation that LLMs struggled with transformations requiring deeper contextual understanding.

Furthermore, our findings regarding prompt engineering reinforce the consensus in recent literature; similar to Tessa et al. [31] and Pandini et al. [32], who found that detailed prompts boost performance for architectural smells, we identified *Chain-of-Thought* as the most effective strategy. This confirms that providing the model with a logical path to follow is more effective than simply providing Context, which we found to be the least robust.

Threats to validity The collected results may not reflect all real-world scenarios, which constitutes a threat to *external validity*. As demonstrated, the LLM refactoring ability varies significantly across specific code smells; therefore, reported results cannot be extrapolated to other smell types. Furthermore, model performance is highly sensitive to the prompting strategy. While we mitigated this by using a set of six different prompt types, these factors inherently limit the extent to which the results represent the models' absolute capabilities. Our decision to alter DeepSeek's prompt to generate outputs may have also introduced bias into the results. Moreover, depending on external code smell detectors for both building the dataset and assessing the LLM-generated code may have introduced a circular validation bias, which we attempted to mitigate by using multiple tools. Also the choice of a specific LLM also affects the accuracy for both csmell detection and refactoring. The analyzed set of LLMs, prompting strategies, and code smells was significantly smaller as compared to all possible scenarios. Therefore, the results could not be directly transferred and generalized. However, they demonstrate trends and limitations that are likely to be present in the similiar settings as well.

Another factor is related to the fact that we used the subject LLMs' as autonomous refactoring agents rather than interactive tools. This differs from the "human-in-the-loop" or "multi-agent" approaches analyzed in recent literature, such as RefactorGPT [20]. Our goal was to test whether general-purpose LLMs could independently identify and refactor code smells, thereby providing assistance to less-experienced developers. Finally, to evaluate models that are commonly available, we limited our analysis to two model families, and this choice may have influenced the data, as highlighted in Sec. IV. Also, StarCoder2 [14] and Code Llama [33], designed explicitly for code generation, might produce different results. To mitigate this issue, each of the four models responded to an average of 1300 [Prompt, Code Smell] random pairs.

Finally, the temporal constraints allowed for running each prompt only once, which does not take into account the stochastic nature of LLMs.

VI. Conclusions

In this study we presented how four commonly available, general-purpose LLMs performed in code smell detection and refactoring of Java programs.

There are two main contributions of the work. First, the subject LLMs detected code smells more accurately if they were explicitly instructed to refactor the code, and the detection was only an intermediate step towards the objective. The effect was better when combined with a Chain-of-Thought prompt, which mandated the model to plan and explain the steps undertaken to complete the task. On the other hand, the prompt composition had only minimal impact on the LLMs' refactoring performance: while **prompts containing more information tend to produce better results**, this advantage was not significant. The second finding is that the model performance varied per type of code smell. Code smells that could be resolved with simpler refactorings, such as LM, achieved better results. In contrast, **refactorings that required a deeper understanding of the code frequently failed and introduced new code smells as a side effect.**. A similar observation concerns the choice of the model: while none of them seemed to clearly dominate others, some of them consistently provided incorrect responses.

Results indicate that the analyzed general-purpose LLMs, without being fine-tuned for code analysis, are not capable of refactoring code smells without human oversight. **They frequently produce code that does not compile and only rarely apply refactorings correctly. Hence, we recommend using such models only as supportive tools to assist informed developers, rather than as independent, unsupervised agents capable of substituting software engineers in refactoring code smells.**

The findings of this study lay the ground for further research in several directions. To address the problem of low

success rates in complex refactorings and the hallucinations observed in this study, future work could incorporate Retrieval-Augmented Generation (RAG) and specialized, refactoring-focused LLMs into the pipeline. This could help to determine whether narrowly specialized training could mitigate the identified risks and the performance gap. Beyond autonomous performance, the *cost of correction* warrants investigation. It remains an open question whether the time required for a developer to fix the code generated by a model outweighs the speed of manual refactoring. Furthermore, the impact likely varies based on developer seniority; therefore, further studies should discriminate between novice and expert developers' outcomes to identify where LLMs' assistance is most viable. Given the stochastic nature of LLMs, the reliability of a *single-pass* approach represents a significant methodological risk. Future experimental designs should explore whether multiple iterations or *self-debugging* cycles, where the model receives feedback on the compile errors, could resolve the issues identified in this study, or if models tend to converge on the same defects regardless of the number of attempts.

The replication package is available in Zenodo[6].

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Science & Business Media, 2006.

[3] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*. IEEE, 2003, pp. 381–384.

[4] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2004, pp. 350–359.

[5] S. Vidal, H. Vazquez, A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi, "Jspirit: a flexible tool for the analysis of code smells," 2015.

[6] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[7] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *2007 IEEE International Conference on Software Maintenance*, 2007.

[8] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 268–278.

[9] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9378-4

[10] F. Palomba, R. Oliveto, and A. De Lucia, "Investigating code smell co-occurrences using association rule learning: A replicated study," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, 2017, pp. 8–13.

[11] B. Walter, F. A. Fontana, and V. Ferme, "Code smells and their collocations: A large-scale experiment on open-source systems," *J. Syst. Softw.*, vol. 144, pp. 1–21, 2018. [Online]. Available: https://doi.org/10.1016/j.jss.2018.05.057

[12] L. L. Silva, J. R. d. Silva, J. E. Montandon, M. Andrade, and M. T. Valente, "Detecting code smells using chatgpt: Initial insights," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024.

[13] J. Cordeiro, S. Noei, and Y. Zou, "Llm-driven code refactoring: Opportunities and limitations," *2025 IEEE/ACM Second IDE Workshop (IDE)*, pp. 32–36, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:277677973

[14] ——, "An empirical study on the code refactoring capability of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2411.02320

[15] D. Wu, F. Mu, L. Shi, Z. Guo, K. Liu, W. Zhuang, Y. Zhong, and L. Zhang, "ismell: Assembling llms with expert toolsets for code smell detection and refactoring," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1345–1357. [Online]. Available: https://doi.org/10.1145/3691620.3695508

[16] G. Shetty and T. Sharma, "Mapping code smells and refactorings accurately: Insights from an empirical study," Preprint, 2025. [Online]. Available: https://tusharma.in/preprints/ESEM2025_Smells_Refactoring_Mapping.pdf

[17] A. R. Sadik and S. Govind, "Benchmarking llm for code smells detection: Openai gpt-4.0 vs deepseek-v3," 2025. [Online]. Available: https://arxiv.org/abs/2504.16027

[18] A. Midolo and E. Tramontana, "Refactoring loops in the era of llms: A comprehensive study," *Future Internet*, vol. 17, no. 9, 2025.

[19] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu, "An empirical study on the potential of llms in automated software refactoring," 2024. [Online]. Available: https://arxiv.org/abs/2411.04444

[20] M. A. Karabiyik, "Refactorgpt: a ChatGPT-based multi-agent framework for automated code refactoring," *PeerJ Computer Science*, vol. 11, 2025. [Online]. Available: https://doi.org/10.7717/peerj-cs.3257

[21] A. Alazba, H. Aljamaan, and M. Alshayeb, "Smellybot: An ai-powered software bot for code smell detection," *Software: Practice and Experience*, 2025.

[22] N. Alomari, A. Alazba, H. Aljamaan, and M. Alshayeb, "Smellycode++: Multi-label dataset for code smell detection," *Scientific Data*, 2025.

[23] N. Anquetil, A. Etien, G. Andreo, and S. Ducasse, "Decomposing god classes at siemens," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019.

[24] A. Kovacevic, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokic, and G. Sladic, 2022.

[25] M. Shahidi, M. Ashtiani, and M. Zakeri, "An automated extract method refactoring approach to correct the long method code smell," *Journal of Systems and Software*, 2022.

[26] D. Yu, Y. Xu, L. Weng, J. Chen, X. Chen, and Q. Yang, "Efficient feature envy detection and refactoring based on graph neural network," *Automated Software Engineering*, vol. 32, 12 2024.

[27] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," 09 2013.

[28] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, "A systematic literature review on the code smells datasets and validation mechanisms," *ACM Computing Surveys*, 2023.

[29] D. Cruz, A. Santana, and E. Figueiredo, "Detecting bad smells with machine learning algorithms: an empirical study," 2020.

[30] K. Prete, N. Rachatasumrit, and M. Kim, "Catalogue of template refactoring rules," *Univ. of Texas at Austin, Tech. Rep. UTAUSTINECE-TR-041610*, 2010.

[31] C. Tessa, M. Bochicchio, and F. Arcelli Fontana, *Exploring Architectural Smells Detection Through LLMs*, ser. Lecture Notes in Computer Science. Springer, 2025, vol. 15929, pp. 90–98. [Online]. Available: https://doi.org/10.1007/978-3-032-02138-0_6

[32] G. Pandini, A. Martini, A. N. Videsjorden, and F. A. Fontana, "An exploratory study on architectural smell refactoring using large languages models," in *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*, 2025.

[33] B. Rozière, Gehring, Gloeckle, Sootla, Gat, Tan, Adi, Liu, Remez, J. Rapin, Kozhevnikov, I. Evtimov, Bitton, Bhatt, Ferrer, A. Grattafiori, Xiong, D'efossez, Copet, Azhar, Touvron, Martin, N. Usunier, Scialom, and Synnaeve, "Code llama: Open foundation models for code," *ArXiv*, vol. abs/2308.12950, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261100919

[6]https://doi.org/10.5281/zenodo.17572881

Microservice Decomposition using Many-Objective Optimization

Vadim Peczyński
Faculty of Electronics, Telecommunication and Informatics
Gdansk University of Technology
Gdansk, Poland
vadim.peczynski@pg.edu.pl

Joanna Szłapczyńska
Faculty of Electronics, Telecommunication and Informatics
Gdansk University of Technology
Gdansk, Poland
joanna.szlapczynska@pg.edu.pl

Abstract— **The literature on Microservices Architecture (MSA) presents numerous approaches for decomposing systems into microservices with the goal of improving quality attributes such as scalability, maintainability, and elasticity. Studies indicate that practitioners typically consider multiple objectives simultaneously when defining service boundaries, making microservice decomposition an inherently multi-objective optimization problem. Within this context, evolutionary algorithms are well suited to exploring trade-offs among competing objectives. This work utilizes the Event Storming graph as the primary input for extracting domain entities and identifying their relationships. By combining semantic analysis of entity names with the structural information embedded in the Event Storming graph, candidate microservice decompositions are generated and optimized using the NSGA-III algorithm. The proposed approach seeks to improve the efficiency and accuracy of deriving microservice boundaries from Event Storming models while providing software architects and decision-makers with a systematic framework for analysing and evaluating decomposition alternatives.**

Keywords- Microservices architecture, MSA, Many-objective optimization, MOO, MSA design

I. INTRODUCTION

The Microservices are built as independently deployable units by fully automated Continuous Integration / Continuous Delivery (CI/CD) pipelines - the centralised management should be kept to bare minimum. They are designed around business capabilities, with each service providing a complete implementation of a specific business function, including the user interface, data persistence layer, and external collaborations. Teams aligned with these business capabilities are cross-functional, bringing together all competencies required for development, such as user experience design, database engineering, and project management [1]. In [2] authors further define Microservice Architecture (MSA) as an architectural style that enables independent deployment, autonomous scaling, and enhanced maintainability of complex software systems.

Architects commonly model both the static structure and the dynamic interactions of a software architecture, including its internal components, to better understand system complexity and external dependencies. Traditionally, modelling approaches such as Business Process Model and Notation (BPMN), Unified Modeling Language (UML), and Domain-Specific Languages (DSLs) have been widely employed for this purpose [3]. However, their adoption in industrial practice has declined, as these approaches often require substantial effort to facilitate effective collaboration and communication with stakeholders beyond the development team. Also in [4], the authors confirm that these methods are becoming increasingly less popular. Intuitive graphical "boxes and lines" approaches are reported to be more widely used for microservices design than UML, with 51% of practitioners preferring them compared to 41.5% for UML. Domain-Specific Languages (DSLs) are even less prevalent, accounting for 27% of usage relative to graphical approaches [4].

According to [5], the primary challenges in migrating to a microservices architecture include identifying appropriate service boundaries and their corresponding bounded contexts, analysing asynchronous communication patterns, and performing event-driven decomposition. As reported in [6], increasing system complexity and scalability challenges are among the primary factors motivating organizations to migrate from microservices back to monolithic architectures, often flaws in the initial service decomposition. Similarly, [7] emphasizes that inappropriate service partitioning can result in substantial technical and operational costs. The selection of suitable decomposition objectives and the partitioning of systems into microservices remain key challenges in the microservices design process [8], [9], [10]. Consequently, evaluating the quality of a microservice decomposition is essential to ensure that services remain sufficiently autonomous and do not become excessively coupled, a condition commonly referred to as "monolithic hell" [11]. To address these challenges, several decomposition strategies have been proposed, including business capability-based decomposition and subdomain-driven decomposition [12]. In parallel, the increasing demand for automated decision support has stimulated the development of numerous approaches, methodologies, and tool-supported frameworks designed to assist software architects in identifying and evaluating appropriate microservice boundaries [13], [14], [15].

To bridge the communication gap between technical teams and business stakeholders and to foster their active involvement

Manuscript received July 07, 2026; revised August 07, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.146

in software architecture design, collaborative and accessible modelling techniques have emerged. One such technique is Event Storming, introduced by Alberto Brandolini. [16]. Event Storming is a workshop-based technique that uses sticky notes to collaboratively model business workflows, processes, and domain knowledge. According to [17], it can be conducted in three forms:

- Big Picture Event Storming, which explores an entire business domain or value stream;
- Process Modelling, which maps current (as-is) and future (to-be) business processes;
- Software Design Event Storming, which translates business processes into specific software and business scenarios.

Based on a set of simple rules, Event Storming is widely recognized as an effective tool for modelling systems developed using Domain-Driven Design (DDD) [12], [18], [19], [20], [21]. During these workshops, stakeholders, domain experts, and developers participate in facilitated collaborative sessions to explore domain knowledge, and establish a shared understanding of the system under development [22].

The aim of this work is to utilize the Event Storming graph as the primary input for extracting domain entities from the system. Event Storming was selected because it captures domain knowledge in a form that enables the identification of bounded contexts, which serve as the basis for constructing the graph and deriving an initial microservice architecture. By combining semantic analysis of entity names with the relationships identified within the Event Storming graph, candidate microservice decompositions are generated using the NSGA-III optimization algorithm. The proposed approach aims to improve the efficiency and accuracy of deriving microservice boundaries from Event Storming models, providing software architects and decision-makers with a systematic means of analysing and evaluating decomposition alternatives.

The remainder of this paper is organized as follows. The Related Work section provides an overview of the existing literature on multi-objective and many-objective optimization approaches for microservice decomposition. The Proposed Method section introduces the semi-automatic approach developed to address the decomposition problem. Next, the Results section presents the outcomes of the experiment conducted using three Event Storming graphs. The Discussion section analyses and interprets the findings, and finally, the Conclusions section summarizes the main contributions of the study and outlines directions for future work.

II. RELATED WORKS

The existing literature presents a wide range of approaches for decomposing systems into microservices with the objective of improving key quality attributes such as maintainability, scalability, and elasticity ([23], [24], [25]). Microservice decomposition has been formulated as a multi-objective optimization problem in several recent studies ([26], [27], [28]), where the goal is to identify service boundaries that minimise inter-service coupling while maximizing intra-service cohesion, alongside satisfying constraints related to service granularity and other architectural quality attributes. Furthermore, empirical evidence indicates that practitioners typically consider multiple competing objectives simultaneously when making decomposition decisions, often balancing at least four criteria during the extraction process [8]. Given the complexity and conflicting nature of these objectives, evolutionary algorithms have emerged as an effective solution, enabling the exploration of large search spaces and the identification of high-quality trade-off solutions.

a) Multi-Objective Evolutionary Optimization

Evolutionary Algorithms (EAs) have proven particularly effective for multi-objective optimization problems due to their population-based search strategy, which enables the simultaneous discovery of multiple Pareto-optimal solutions in a single execution [29]. Multi-Objective Evolutionary Algorithm (MOEA) effectiveness is grounded in key concepts such as Pareto dominance, convergence toward the Pareto front, and the preservation of solution diversity across the set of non-dominated alternatives. Early work in this area, such as the Vector Evaluated Genetic Algorithm (VEGA) [30], established the foundations of multi-objective evolutionary optimization; however, it is generally considered less effective for addressing complex problems characterized by large and high-dimensional search spaces. Subsequent algorithms, including NSGA-II and NSGA-III, introduced significant improvements through elitist selection mechanisms and advanced diversity-preservation strategies ([29], [31]). NSGA-II achieves computational efficiency through fast non-dominated sorting and crowded-comparison operators, eliminating the need for fitness-sharing parameters and contributing to its widespread adoption. Building upon these advances, NSGA-III extends the applicability of evolutionary optimization to many-objective problems by incorporating adaptive normalization and a reference-point-based niching mechanism, enabling a more effective distribution of solutions across high-dimensional Pareto fronts [31].

Other prominent multi-objective evolutionary algorithms include the Strength Pareto Evolutionary Algorithm (SPEA) and its successor, SPEA2, both of which utilize an external archive to preserve non-dominated solutions and a strength-based fitness assignment mechanism to evaluate individual quality [32]. SPEA2 further enhances the original approach by introducing an improved density estimation technique and a refined archive truncation procedure, enabling a more accurate assessment of solution diversity and a better preservation of boundary solutions along the Pareto front. These improvements contribute to a more balanced distribution of solutions and improved convergence toward high-quality trade-off alternatives.

Multi-objective Evolutionary Algorithm based on Decomposition (MOEA/D) offers an alternative strategy for multi-objective optimization by decomposing a problem into a

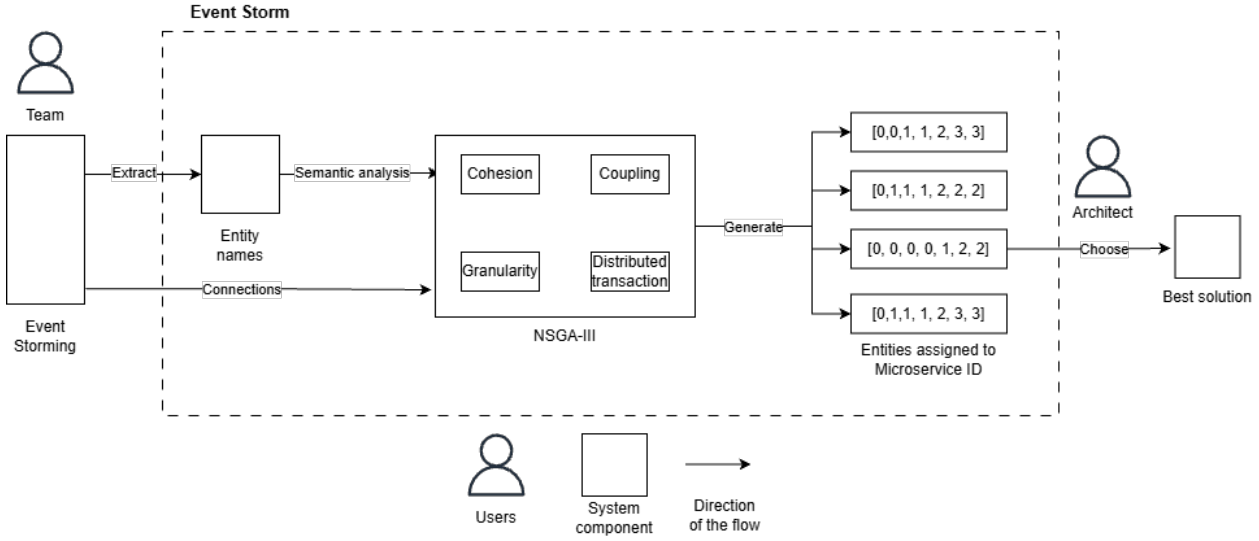


Fig. 1 Overview of the approach for MS decomposition

set of scalar optimization subproblems, each associated with a predefined weight vector [33]. This decomposition enables neighbouring subproblems to share information during the search process, improving computational efficiency and scalability. Despite these advantages, MOEA/D may encounter difficulties when the Pareto front is highly irregular or unevenly distributed, which can limit its ability to maintain a well-spread set of solutions [31].

In contrast to dominance-based approaches, the Indicator-Based Evolutionary Algorithm (IBEA) employs quality indicators to guide the selection process, enabling the direct assessment of solution quality within the objective space [34]. Common variants, such as IBEAε+ and hypervolume-based IBEA (IHD), provide a flexible framework for incorporating decision-maker preferences while eliminating the need for explicit diversity-preservation mechanisms. Fitness values are assigned by evaluating each solution relative to all other individuals in the population, allowing selection pressure to be driven by the chosen performance indicator. Experimental results reported in [34] demonstrate that IBEA consistently outperforms established algorithms such as NSGA-II and SPEA2 across a wide range of benchmark problems, exhibiting strong convergence properties, greater generality, and improved scalability with respect to population size.

b) Algorithms used for Microservice Decomposition

Among the evolutionary algorithms applied to the microservice decomposition problem, NSGA-II is one of the most widely adopted due to its ability to generate diverse sets of Pareto-optimal decomposition alternatives. Its effectiveness in balancing competing architectural objectives has led to its application in a variety of contexts, including cloud migration scenarios, where optimization must account for constraints such as network communication overhead and hosting costs [15], [23], [35]. To address scalability issues associated with large and complex systems, [36] proposes MSExplorer, which reduces the search space through an initial clustering phase prior to performing NSGA-II-based optimization. Beyond microservice decomposition, recent studies have addressed deployment and post-design challenges. Specifically, [37] proposes Yonga, an adaptive placement strategy for resource-constrained environments, while [38] introduces a feature model extraction approach to improve reuse, variability management, and configurability in microservice-based software product lines.

NSGA-III, the successor to NSGA-II, has been applied to complex software decomposition problems involving a larger number of optimization objectives, with implementations reported in [14], [23]. Its reference-point-based selection mechanism makes it particularly suitable for many-objective optimization. Similarly, [39] modelling microservice identification as a many-objective problem and showing that simultaneously optimizing coupling, cohesion, feature modularization, reuse, and network overhead produce higher-quality architectural solutions.

The Indicator-Based Evolutionary Algorithm (IBEA) has also been successfully applied to the microservice decomposition problem through the MSExtractor framework [23]. The reported results indicate that IBEA consistently outperforms NSGA-II and SPEA2 by generating decomposition solutions that provide a more favourable balance between coupling, cohesion, and service granularity. Beyond traditional evolutionary algorithms, hybrid approaches combining Genetic Algorithms with Fuzzy Cognitive Maps have been proposed to capture uncertainty and complex dependencies during decomposition [27], [40]. More recently, researchers have explored combinatorial optimization for automated microservice synthesis [41] and deep reinforcement

learning techniques to derive optimal microservice architectures [35]. In the domain of Software-as-a-Service (SaaS) systems, [33] highlights the importance of variability management through the SAMIM approach, which applies MOEA/D to optimize trade-offs among component commonality, service granularity, and alignment with business objectives.

The decision to create Microservices Architecture (MSA) system is inherently complex and often requires support from dedicated decision-making frameworks and tools. Approaches such as those proposed in [24] facilitate evidence-based decision making by leveraging objective system data, while Multi-Layer Fuzzy Cognitive Maps (MLFCMs) enable scenario analysis and improve the transparency and interpretability of the decision process [27]. The quality of decomposition results is also highly influenced by the configuration of optimization algorithms, making hyperparameter tuning an important factor in achieving effective solutions [42]. Furthermore, weighted loss functions provide a mechanism for incorporating business priorities directly into the optimization process, allowing decomposition outcomes to better align with organizational goals [43]. Given the dynamic nature of software systems, implementation should be viewed as an iterative process that requires continuous monitoring and architectural refinement in response to evolving business requirements and technical constraints [44], [45].

III. Proposed Framework – Event Storm

This section introduces the proposed framework, Event Storm, and subsequently describes the problem formulation and the modifications made to adapt the multi-objective optimization algorithm to the microservice decomposition task. Event Storm utilizes the Event Storming graph to extract domain entities and identify bounded contexts that define an initial microservice architecture. Candidate microservice decompositions are generated by combining semantic analysis of entity names with graph relationships using the NSGA-III optimization algorithm, providing systematic support for evaluating decomposition alternatives.

A. Overview of the method

Event Storming facilitates collaboration between technical and non-technical stakeholders, enabling the collaborative elicitation of domain knowledge. The resulting Event Storming model provides architects with information about domain entities, aggregates, and bounded contexts. Building on this foundation, we extend the Event Storming process by enabling the automated extraction of microservices from an Event Storming graph. This task is formulated as a combinatorial optimization problem, in which entities identified within the graph are partitioned into candidate microservices. Given the large number of possible decompositions, the problem can be modelled as a graph partitioning task, which is known to be NP – hard problem [14], [28]. Consequently, a meta-heuristic search strategy is employed to identify near-optimal solutions within a reasonable computational time. As illustrated in Fig. 1, the proposed Event Storm approach utilizes NSGA-III to explore the search space and generate alternative sets of microservice candidates.

B. Problem formulation

In this work, a microservice architecture $M$is defined as a decomposition of a set of entities E into a collection of microservices, where each microservice represents a cohesive container of related entities. Let $E = \{e_1, e_2, \dots, e_n\}$ denote the set of identified entities in a software system, where n is the total number of entities. The set of candidate microservices is represented as $M = \{m_1, m_2, \dots, m_k\}$, where k denotes the number of microservices and each microservice is assigned a unique identifier. The size of a microservice, denoted as $size(m)$, is defined as the number of entities assigned to it.

A decomposition solution is encoded as a vector of decision variables $X = \{x_1, x_2, \dots, x_n\}$, where $x_i = j$indicates that entity $e_i$is assigned to microservice m_j. For example, the solution $X = \{1,1,2,2,3,3,4,4\}$ represents a decomposition of eight entities into four microservices. Since entities interact to support the required system functionality, dependencies naturally exist among them. Therefore, an effective decomposition should maximize cohesion within microservices while minimizing inter-service coupling and the number of distributed transactions. To support these objectives, the proposed Event Storm approach employs structural and semantic similarity measures to quantify dependencies among entities, guiding the optimization process toward highly cohesive and loosely coupled microservice boundaries.

a) Structural similarity

Each entity $e_i \in E = \{e_1, e_2, \dots, e_n\}$is represented by a structural feature vector $\mathbf{s}_i = [s_{i1}, s_{i2}, \dots, s_{in}]$, where each element s_{ij}denotes the number of structural interactions (connections) between entity $e_i$and entity e_j. In this representation, the vector encodes the connectivity profile of an entity within the system, capturing how it interacts with all other entities in the system. For instance, a vector such as $[0, 0, 0, 4, 0, 0, 0, 1, 4, 1, 0, 1, 2]$indicates that the corresponding entity has four interactions with e_4, one interaction with e_8, four interactions with e_9, and additional interactions with other entities as specified by non-zero entries. Based on these structural representations, the structural similarity between two entities can be calculated:

$$sim_{str}(e_i, e_j) = \sum_{r \in R} \frac{N_r(e_i, e_j)\text{-}min(N_r)}{max(N_r)\text{-}min(N_r)} \in [0,1]$$

where:

- $N_r(e_i, e_j)$ is the number of times entities $e_i$and $e_j$appear as connected elements in process r
- R is the set of all processes,

This approach enables the identification of entities that exhibit similar structural roles, even if they are not directly connected, thereby supporting the detection of cohesive groups during the microservice decomposition process.

b) Semantic similarity

Semantic similarity captures conceptual and domain-level relationships between entities and is computed using vector representations derived from spaCy embeddings. Each entity e_i is represented by a semantic vector $\mathbf{s}_i$, and the similarity between two entities is defined using cosine similarity:

$$\mathrm{sim}_{sem}(e_i, e_j) = \frac{\mathbf{s}_i \cdot \mathbf{s}_j}{\| \mathbf{s}_i \| \ \| \mathbf{s}_j \|} \in [0,1],$$

where ·denotes the dot product and ‖·‖is the Euclidean norm. This measure reflects the conceptual proximity of entities in the embedding space, enabling the capture of domain-level similarity beyond exact lexical matches.

c) Overall similarity

The overall similarity between two entities combines both structural and semantic aspects. It is defined as the sum of string-based similarity and semantic similarity:

$$\mathrm{sim}(e_i, e_j) = 0.5 * \mathrm{sim}_{str}(e_i, e_j) + 0.5 * \mathrm{sim}_{sem}(e_i, e_j) \in [0,1]$$

This aggregated measure integrates lexical overlap with embedding-based semantic relationships, providing a more comprehensive similarity assessment for entity comparison.

C. MOEA adoption

The Event Storm evaluates a candidate microservice decomposition using a multi-objective fitness function that assesses structural quality, communication cost, and architectural balance. Each solution is encoded as a vector $X = \{x_1, x_2, \dots, x_n\}$, where each element x_i assigns entity e_i to a specific microservice. The evaluation procedure operates on a normalized representation of this assignment and computes four complementary objectives: cohesion, coupling, granularity, and distributed transaction cost.

1) Solution representation

Given an input solution X, the algorithm first applies a normalization step to ensure consistent labeling of microservices. The normalized vector X' induces a partition of entities into a set of clusters $M = \{m_1, m_2, \dots, m_k\}$, where k denotes the number of distinct microservices in the solution.

2) Initial population

The initial population is constructed using a random initialization strategy. Given a predefined maximum number of microservices (n), each entity in the application is randomly assigned to a microservice, resulting in a set of candidate decompositions. To ensure the generation of meaningful service boundaries, a constraint parameter (minSize) is introduced, representing the minimum number of classes permitted within a microservice. This constraint prevents the creation of undersized services and contributes to the generation of more balanced and practically viable decomposition solutions.

3) Objective functions

The quality of a candidate microservice decomposition is evaluated through a multi-objective fitness function that incorporates both optimization objectives and feasibility constraints. This fitness function provides a quantitative measure for assessing and comparing alternative decomposition solutions. Each objective corresponds to a distinct architectural quality attribute and is formulated to be either minimized or maximized according to the desired system characteristics. Consequently, the search process aims to identify solutions that achieve an optimal balance among potentially conflicting objectives while adhering to all specified constraints. In this work, the optimization framework focuses on the following four objectives:

a) Cohesion (maximize)

Cohesion measures the degree of functional and structural relatedness of entities within the same microservice. It is computed using a cohesion function defined for each microservice, following the formulation in [28] and adapted to measure entity similarity:

$$Coh(m) = 1 - \frac{\sum_{\substack{e_i, e_j \in m \\ e_i \neq e_j}} sim(e_i, e_j)}{\frac{| m | (| m | - 1)}{2}}$$

where:

- $sim(e_i, e_j)$ is the similarity between entities e_i and e_j,
- $| m |$ is the number of entities in microservice m.

Then for overall cohesion:

$$Cohesion(\mathcal{M}) = 1 - \frac{\sum_{m_i \in \mathcal{M}} Coh\,(m_i)}{| \mathcal{M} |}$$

where:

- $Coh(m_i)$ is the cohesion value of microservice m_i,
- $| \mathcal{M} |$ is the total number of microservices.

Higher cohesion values indicate stronger intra-service relationships. In the optimization process, cohesion is maximized.

b) Coupling (minimize)

Coupling quantifies the level of interdependencies between different microservices. It is computed following the formulation in [28], with adaptations to account for entity similarity, as follows:

$$Cou(m_1, m_2) = \frac{\sum_{e_i \in m_1,\, e_j \in m_2} sim(e_i, e_j)}{| m_1 | \cdot | m_2 |}$$

where:

- $sim(e_i, e_j)$is the similarity between entities $e_i$and e_j,
- $| m_1 |$and $| m_2 |$denote the sizes of the two microservices.

Lower coupling values are preferred, as they indicate reduced inter-service communication and dependency.

c) Granularity (maximize)

Granularity captures the decomposition balance in terms of the number of generated microservices relative to the number of entities. The objective function is a normalized and modified version of the objective function proposed in [28], formulated as follows::

$$G(X') = \frac{k}{| E |}$$

where:

- $k$is the number of detected microservices and
- $| E |$is the total number of entities.

This objective promotes controlled decomposition and avoids extreme fragmentation.

d) Distributed Transactions Cost (minimize)

Distributed transaction cost evaluates the communication overhead induced by inter-service interactions. It is computed based on the execution paths or dependency graph PE:

$$DT_{cost} = \frac{\sum_{r \in R}^{\square} Jump\left(a_{u_i} \neq a_{u_{i+1}}\right)}{\sum_{r \in R}(| U_r | - 1)}$$

where:

- a_{u_i} is microservice associated with entity u_i
- $\mathbf{Jump}(A) = \begin{cases} 1, & \textit{if A is true} \\ 0, & \textit{otherwise} \end{cases}$
- $| U_r |$= number of entities in process r
- R is the set of all processes

This objective penalizes decompositions that require frequent cross-service transactions.

4) Constraints

To ensure meaningful microservice formation, a minimum size constraint is enforced for each cluster:

$$\min_{m_i \in M} | m_i | \geq \text{min\_size}$$

If any microservice violates this constraint, the solution is penalized accordingly:

$$G_{constraint}(X') = \text{min\_size} - \min\left(| m_1 |, | m_2 |, \ldots, | m_k |\right)$$

5) Multi-objective Optimization Formulation

The final fitness vector is defined as:

$$F(X') = [-C_{oh}(X'), C_{ou}(X'), DT(X'), -G(X')]$$

The optimization problem simultaneously minimizes all objectives, encouraging decompositions that exhibit high cohesion, low coupling, low transaction cost, and balanced granularity. In the PyMoo framework, all objective functions are formulated as minimization problems. Consequently, cohesion and coupling metrics are multiplied by −1 to ensure consistency with this minimization requirement.

IV. Experiments & Results

This section presents the experimental evaluation of the proposed approach for automated microservice decomposition based on Event Storming graphs. The study investigates how different architectural preference profiles influence the resulting decompositions and assesses the stability and quality of the generated solutions under repeated optimization runs.

a) Experiment

The proposed approach was evaluated using three Event Storming graphs representing different software systems. The experiment involved two hypothetical software architects with contrasting design preferences, whose objective configurations were used to guide the optimization process. The first architect represents a monolithic-oriented perspective, prioritizing cohesion and distributed transaction cost, thereby favoring more strongly integrated services. The second architect represents a microservice-oriented perspective, emphasizing coupling and granularity to promote loosely coupled and finer-grained services. For each Event Storming graph, the optimization process was executed 15 times to account for the stochastic nature of the NSGA-III algorithm.

b) Results

The resulting Pareto-optimal solutions were ranked using the TOPSIS method to select the most preferred decomposition in each run. This enables a consistent selection of representative solutions from the Pareto front for subsequent analysis. For one of the evaluated Event Storming graphs, the mean objective values of the selected solutions illustrate the effect of the differing preference profiles. The monolithic-oriented configuration achieved higher cohesion reflecting a stronger focus on intra-service relationships. In contrast, the microservice-oriented configuration resulted in lower coupling, higher granularity, and reduced distributed transaction cost, indicating a shift towards more independent and fine-grained services.

The quality of the obtained Pareto fronts is assessed using the hypervolume (HV) indicator, which measures the volume

of the objective space dominated by the Pareto set relative to a predefined reference point. In this study, all objectives are normalized to the range $[0, 1]$, and the reference point is defined as $r = (1.1, 1.1, 1.1, 1.1)$. The hypervolume is therefore computed with respect to this reference point, where higher values indicate better convergence towards the Pareto-optimal region and improved diversity of solutions.

The maximum achievable hypervolume corresponds to the ideal case in which the Pareto front reaches the origin $(0, 0, 0, 0)$, yielding $HV_{\max} = (1.1 - 0)^4 = 1.1^4 = 1.4641$. Accordingly, the valid range of the hypervolume indicator is $0 \leq HV \leq 1.4641$. In the conducted experiments, a hypervolume value of approximately 0.95 was obtained, which corresponds to about 68% of the theoretical maximum. This indicates that the proposed method is robust with respect to varying architectural priorities, while still enabling controlled steering of the decomposition outcome according to stakeholder preferences. The results are presented in Table 1.

TABLE I. RESULTS OF THE EXPERIMENTS

Scenario	Results		
	Metric	***User 1 (Monolithic)***	***User 2 (Microservice)***
1	Cohesion (mean)	0.2366	0.0336
	Coupling (mean)	0.0799	0.0901
	Granularity (mean)	0.3333	0.9000
	DT Cost (mean)	0.1334	0.7619
	Hypervolume (mean)	0.9646	
2	Cohesion (mean)	0.3045	0.04774
	Coupling (mean)	0.1582	0.16584
	Granularity (mean)	0.2778	0.87778
	DT Cost (mean)	0.3281	0.92982
	Hypervolume (mean)	0.9541	
3	Cohesion (mean)	0.2532	0.0615
	Coupling (mean)	0	0.2319
	Granularity (mean)	0.1667	0.8476
	DT Cost (mean)	0	0.9111
	Hypervolume (mean)	1.0534	

V. DISCUSSION

The following section describe the key design decisions underpinning the proposed approach and provide a discussion of the experimental results. Presented framework for microservice extraction operates on the Event Storming graph. Cards are grouped at the entity level, although a finer-grained (card-level) decomposition is also possible. The quality of the resulting microservice decomposition is strongly dependent on the quality of the input Event Storming graph. Therefore, a

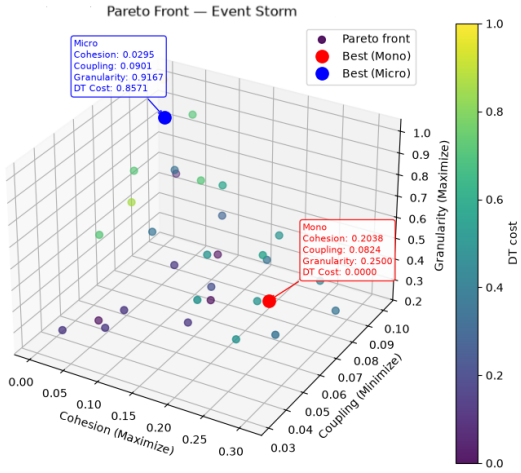


Fig. 2 Representative results for User 1 (red) and User 2 (blue) for Scenario 1

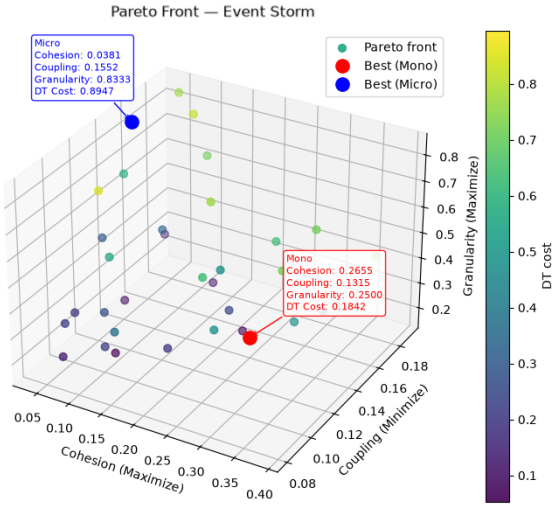


Fig. 3 Representative results for User 1 (red) and User 2 (blue) for Scenario 2

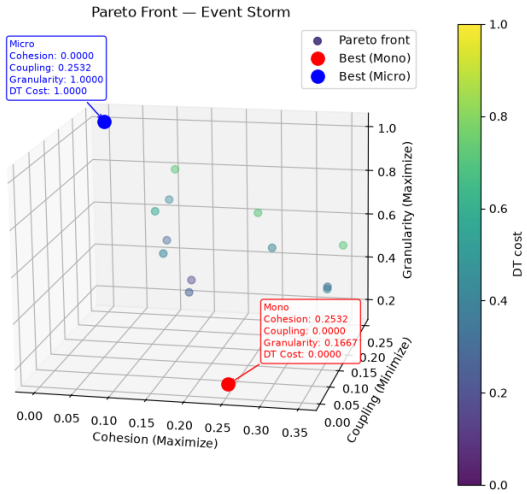


Fig. 4 Representative results for User 1 (red) and User 2 (blue) for Scenario 3

preliminary refinement phase is required, including the simplification of business processes, as well as the removal of redundant or unused cards, to ensure a consistent and high-quality representation of the domain. This step is essential, as microservices are expected to be small, cohesive, and functionally meaningful units.

A multi-objective formulation is necessary, as relying solely on coupling may lead to overly fine-grained decompositions (e.g., single-entity services), whereas optimizing only cohesion may collapse the solution into a single monolithic service. Initial experiments further indicated that these objectives alone tend to produce extreme solutions. To address this limitation, additional objectives related to granularity and distributed transaction cost are introduced to better control service size and enforce user-defined constraints.

The goal of the approach is to group entities that are both structurally and semantically related into candidate microservices. The evolutionary optimization process progressively favors solutions that satisfy these objectives, while weaker candidates characterized by low cohesion or high distributed transaction cost are eliminated over successive generations. Nevertheless, the generation of a deployable microservice architecture requires human involvement for Event Storming preparation, validation, and post-processing refactoring, which is outside the scope of this work.

The inter-entity dependencies are inferred using structural and semantic similarity measures, which act as heuristics for capturing latent relationships in the absence of explicit use-case information. While this enables automation in incomplete or informal modeling settings, it also introduces the risk of grouping entities that are syntactically or semantically similar but not necessarily functionally related.

The results in Table 1 demonstrate that the proposed optimization framework successfully adapts the generated decompositions to different stakeholder preferences while maintaining comparable optimization quality.

Across all scenarios, the monolithic-oriented configuration consistently produced substantially higher cohesion than the microservice-oriented configuration. In contrast, the microservice-oriented configuration achieved significantly higher granularity, indicating the generation of smaller and more fine-grained microservices. This behavior is consistent with the optimization objectives assigned to each preference profile.

The remaining objectives further illustrate the trade-offs between the two architectural styles. In Scenarios 1 and 2, coupling values are comparable for both configurations, with the monolithic-oriented profile producing slightly lower values. In Scenario 3, however, the monolithic-oriented configuration achieved zero coupling and distributed transaction cost, indicating a decomposition with no inter-service communication. The microservice-oriented configuration exhibited higher coupling (0.2319) and distributed transaction cost (0.9111), reflecting the increased communication overhead typically associated with finer-grained service decomposition.

The hypervolume values remained consistently high across all scenarios, ranging from 0.9541 to 1.0534. This indicates that the proposed NSGA-III-based optimization approach produced well-converged and diverse Pareto fronts regardless of the selected architectural preference profile. Overall, the results confirm that modifying the optimization objectives effectively steers the decomposition process toward either monolithic-oriented or microservice-oriented solutions while maintaining high optimization quality. Figures 1–3 present representative optimization results for the three experimental scenarios. The figures illustrate a consistent distinction between the proposed solutions generated for User 1 and User 2, reflecting the different architectural preference profiles across all experiments.

VI. Threats to validity

Threats to internal validity relate to factors within the study that may influence the results. Due to the stochastic nature of the proposed evolutionary approach, some variability in outcomes is expected. To mitigate this, each experiment was repeated 15 times for each user, and mean values were used for analysis. The comparison relied on the Hypervolume (HV) indicator, which, while widely used due to its Pareto compliance, may not fully capture all aspects of performance when used in isolation. Moreover, without knowledge of the true Pareto Front, convergence metrics such as Generational Distance (GD) and Inverted Generational Distance (IGD) cannot be reliably computed. As any approximation would introduce bias, HV was selected as the primary indicator; however, future work should incorporate additional metrics to better capture convergence and diversity.

Threats to construct validity arise from the chosen evaluation metrics. Although cohesion, coupling, granularity, and distributed transaction cost provide multiple perspectives on decomposition quality, they may not capture all relevant aspects. Future studies could integrate additional quality measures to improve evaluation comprehensiveness.

Threats to external validity are mainly related to limited generalizability. The evaluation was conducted on three Event Storming graphs, which constrains broader applicability. Additionally, qualitative assessment was performed only by the authors, without input from external experts. Future work will involve practitioner studies and comparative evaluations with existing microservice decomposition approaches to strengthen external validity.

VII. Conclusions

In this paper, we presented Event Storm, a novel framework for microservice extraction that formulates the decomposition problem as a multi-objective combinatorial optimization task. The proposed method leverages the NSGA-III algorithm to search for optimal decompositions of Event Storming graphs

while simultaneously considering both structural and semantic relationships among identified entities. The evaluation results demonstrate that Event Storm is capable of generating microservice decompositions characterized by high cohesion and low coupling, two key quality attributes of effective microservice architectures.

As part of our future work, we plan to investigate alternative optimization algorithms and compare the proposed approach with existing state-of-the-art microservice decomposition techniques. We also intend to conduct a more extensive evaluation involving software architects and developers, as well as a broader set of case studies, to further assess the practical applicability of the approach. In addition, future research will incorporate non-functional quality attributes that play a critical role in microservice architectures, including scalability, cost efficiency, and testability, thereby enabling a more comprehensive assessment of decomposition quality.

Furthermore, we plan to explore preference-based multi-objective optimization techniques, particularly the integration of the w-dominance relation[46] into the decomposition process. Unlike traditional Pareto dominance, w-dominance enables the incorporation of decision-maker preferences and associated uncertainties through weight intervals that express the relative importance of optimization objectives. This approach would allow software architects to explicitly prioritize architectural concerns according to project-specific requirements without requiring prior knowledge of expected solutions. By guiding the search process toward solutions that better reflect stakeholder preferences, w-dominance has the potential to improve the relevance and practical usefulness of the generated microservice decomposition alternatives.

DATA AVAILABILITY

Data will be made available on request.

REFERENCES

[1] J. Lewis and M. Fowler, "Microservices - a definition of this new architectural term," 2014.

[2] L. Qian, J. Li, X. He, R. Gu, J. Shao, and Y. Lu, "Microservice extraction using graph deep clustering based on dual view fusion," *Inf. Softw. Technol.*, vol. 158, p. 107171, 2023, doi: https://doi.org/10.1016/j.infsof.2023.107171.

[3] O. Zimmermann, K. Luban, M. Stocker, and G. Bernard, "Continuous process model refinement from business vision to event simulation and software automation," in *Proceedings of the 5th International Workshop on Software-intensive Business: Towards Sustainable Software Business*, ACM, May 2022, pp. 59–66. doi: 10.1145/3524614.3528631.

[4] M. Waseem, P. Liang, M. Shahin, A. Di Salle, and G. Márquez, "Design, monitoring, and testing of microservices systems: The practitioners' perspective," *Journal of Systems and Software*, vol. 182, p. 111061, 2021, doi: https://doi.org/10.1016/j.jss.2021.111061.

[5] H. Ünlü, D. E. Kennouche, G. K. Soylu, and O. Demirörs, "Microservice-based projects in agile world: A structured interview," *Inf. Softw. Technol.*, vol. 165, p. 107334, 2024, doi: https://doi.org/10.1016/j.infsof.2023.107334.

[6] R. Su, X. Li, and D. Taibi, "Back to the Future: From Microservice to Monolith," May 2023, doi: 10.48550/arXiv.2308.15281.

[7] Y. Wei, Y. Yu, M. Pan, and T. Zhang, "A Feature Table approach to decomposing monolithic applications into microservices," *ACM International Conference Proceeding Series*, pp. 21–30, May 2020, doi: 10.1145/3457913.3457939.

[8] L. Carvalho, A. Garcia, W. K. G. Assunção, R. de Mello, and M. de Lima, "Analysis of the Criteria Adopted in Industry to Extract Microservices," in *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, May 2019, pp. 22–29. doi: 10.1109/CESSER-IP.2019.00012.

[9] T. Lopes and A. R. Silva, "Monolith Microservices Identification: Towards An Extensible Multiple Strategy Tool," in *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, Mar. 2023, pp. 111–115. doi: 10.1109/ICSA-C57050.2023.00034.

[10] V. Peczyński, J. Szłapczyńska, and A. Szopińska, "Patterns in Design of Microservices Architecture: IT Practitioners' Perspective," *WiPiEC Journal - Works in Progress in Embedded Computing Journal*, vol. 11, no. 1, p. 10, Sep. 2025, doi: 10.64552/wipiec.v11i1.101.

[11] S. Santos and A. R. Silva, "Microservices Identification in Monolith Systems: Functionality Redesign Complexity and Evaluation of Similarity Measures," *Journal of Web Engineering*, vol. 21, no. 5, pp. 1543–1582, May 2022, doi: 10.13052/jwe1540-9589.2158.

[12] C. Richardson, *Microservices Patterns: With examples in Java*. Manning, 2018. [Online]. Available: https://books.google.pl/books?id=UeK1swEACAAJ

[13] D. Bajaj, U. Bharti, I. Gupta, P. Gupta, and A. Yadav, "GTMicro—microservice identification approach based on deep NLP transformer model for greenfield developments," *International Journal of Information Technology (Singapore)*, vol. 16, no. 5, pp. 2751–2761, Jun. 2024, doi: 10.1007/S41870-024-01766-5.

[14] T. Kinoshita and H. Kanuka, "Enhancing Automated Microservice Decomposition via Multi-Objective Optimization," *IEEE Access*, vol. 12, pp. 55697–55710, Apr. 2024, doi: 10.1109/ACCESS.2024.3389700.

[15] K. Sellami, M. A. Saied, and A. Ouni, "A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications," in *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, in EASE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 201–210. doi: 10.1145/3530019.3530040.

[16] A. Brandolini, "Introducing Event Storming," 2013.

[17] E. van Kelle, G. Verschatse, and K. Baas-Schwegler, *Collaborative Software Design: How to facilitate domain modeling decisions*. Simon and Schuster, 2025.

[18] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated, 2019. [Online]. Available: https://books.google.pl/books?id=iul3wQEACAAJ

[19] V. Vernon, *Domain-driven Design Distilled*. Addison-Wesley, 2016. [Online]. Available: https://books.google.pl/books?id=h0u7jwEACAAJ

[20] V. Khononov, *Learning Domain-Driven Design*. O'Reilly Media, Inc., 2021.

[21] S. R. Goniwada, *Cloud Native Architecture and Design*. Apress, 2022. doi: 10.1007/978-1-4842-7226-8.

[22] H. Gardner, A. F. Blackwell, and L. Church, "The patterns of user experience for sticky-note diagrams in software requirements workshops," *J. Comput. Lang.*, vol. 61, p. 100997, 2020, doi: https://doi.org/10.1016/j.cola.2020.100997.

[23] I. Saidani, A. Ouni, M. W. Mkaouer, and A. Saied, "Towards Automated Microservices Extraction Using Muti-objective Evolutionary Search," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11895 LNCS, SPRINGER INTERNATIONAL PUBLISHING AG, 2019, pp. 58–63. doi: 10.1007/978-3-030-33702-5_5.

[24] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to Microservices: An assessment framework," *Inf. Softw. Technol.*, vol. 137, May 2021, doi: 10.1016/j.infsof.2021.106600.

[25] T. Kinoshita and H. Kanuka, "Automated Microservice Decomposition Method as Multi-Objective Optimization," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, Mar. 2022, pp. 112–115. doi: 10.1109/ICSA-C54293.2022.00028.

[26] A. Christoforou, L. Odysseos, and A. S. Andreou, "Migration of software components to microservices: Matching and synthesis," in *ENASE 2019 - Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, E. Damiani, G. Spanoudakis, and L. Maciaszek, Eds., SCITEPRESS - Science and Technology Publications, 2019, pp. 134–146. doi: 10.5220/0007732101340146.

[27] A. Christoforou, A. S. Andreou, M. Garriga, and L. Baresi, "Adopting microservice architecture: A decision support model based on genetically evolved multi-layer FCM," *Appl. Soft Comput.*, vol. 114, p. 108066, May 2022, doi: 10.1016/j.asoc.2021.108066.

[28] K. Sellami, A. Ouni, M. A. Saied, S. Bouktif, and M. W. Mkaouer, "Improving microservices extraction using evolutionary search," *Inf. Softw. Technol.*, vol. 151, p. 106996, 2022, doi: https://doi.org/10.1016/j.infsof.2022.106996.

[29] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr. 2002, doi: 10.1109/4235.996017.

[30] J. D. Schaffer, "Some Experiments in Machine Learning Using Vector Evaluated Genetic Algorithms," Vanderbilt University, 1985.

[31] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, Part I: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2014, doi: 10.1109/TEVC.2013.2281535.

[32] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm," 2001. doi: https://doi.org/10.3929/ethz-a-004284029.

[33] Q. Zhang and H. Li, "MOEA/D: A multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 6, pp. 712–731, May 2007, doi: 10.1109/TEVC.2007.892759.

[34] E. Zitzler and S. Künzli, "Indicator-Based Selection in Multiobjective Search," in *Parallel Problem Solving from Nature - PPSN VIII*, E. K. and L. J. A. and S. J. and M.-G. J. J. and B. J. A. and R. J. E. and T. P. and K. A. and S. H.-P. Y. X. and Burke, Ed., Springer Berlin Heidelberg, 2004, pp. 832–842. doi: 10.1007/978-3-540-30217-9_84.

[35] K.-H. Chow, U. Deshpande, V. Deenadayalan, S. Seshadri, and L. Liu, "Atlas: Hybrid Cloud Migration Advisor for Interactive Microservices," in *EuroSys 2024 - Proceedings of the 2024 European Conference on Computer Systems*, ACM, May 2024, pp. 870–887. doi: 10.1145/3627703.3629587.

[36] S. Izadpanah, A. Rasoolzadegan, S. Abrishami, and A. Mousavi, "Improving Microservices Identification for Migration to Cloud-Native Applications," *IEEE Trans. Serv. Comput.*, pp. 1–15, Apr. 2026, doi: 10.1109/TSC.2026.3658420.

[37] A. Mwotil, B. Kanagwa, and E. Bainomugisha, "Yonga: An Adaptive Metaheuristic-Based Microservice Placement Approach for Low-Resource Distributed Systems," *IEEE Access*, vol. 14, pp. 6647–6663, Apr. 2026, doi: 10.1109/ACCESS.2026.3653120.

[38] W. D. F. Mendonça, W. K. G. Assunção, L. V Estanislau, S. R. Vergilio, and A. Garcia, "Towards a Microservices-Based Product Line with Multi-Objective Evolutionary Algorithms," in *2020 IEEE Congress on Evolutionary Computation (CEC)*, Jul. 2020, pp. 1–8. doi: 10.1109/CEC48606.2020.9185776.

[39] W. K. G. Assunção *et al.*, "Analysis of a many-objective optimization approach for identifying microservices from legacy systems," *Empir. Softw. Eng.*, vol. 27, no. 2, Mar. 2022, doi: 10.1007/S10664-021-10049-7.

[40] M. Abdellatif *et al.*, "A taxonomy of service identification approaches for legacy software systems modernization," *Journal of Systems and Software*, vol. 173, May 2021, doi: 10.1016/j.jss.2020.110868.

[41] G. Filippone, M. Autili, F. Rossi, and M. Tivoli, "Migration of Monoliths through the Synthesis of Microservices using Combinatorial Optimization," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2021, pp. 144–147. doi: 10.1109/ISSREW53611.2021.00056.

[42] R. Yedida, R. Krishna, A. Kalia, T. Menzies, J. Xiao, and M. Vukovic, "Lessons learned from hyper-parameter tuning for microservice candidate identification," in *Proceedings - 2021 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021*, IEEE, May 2021, pp. 1141–1145. doi: 10.1109/ASE51524.2021.9678704.

[43] R. Yedida, R. Krishna, A. Kalia, T. Menzies, J. Xiao, and M. Vukovic, "An expert system for redesigning software for cloud applications," *Expert Syst. Appl.*, vol. 219, p. 119673, May 2023, doi: 10.1016/j.eswa.2023.119673.

[44] M. Camilli, C. Colarusso, B. Russo, and E. Zimeo, "Domain Metric Driven Decomposition of Data-Intensive Applications," in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2020, pp. 189–196. doi: 10.1109/ISSREW51248.2020.00071.

[45] M. Camilli, C. Colarusso, B. Russo, and E. Zimeo, "Actor-Driven Decomposition of Microservices through Multi-level Scalability Assessment," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, May 2023, doi: 10.1145/3583563.

[46] R. Szlapczynski and J. Szlapczynska, "W-dominance: Tradeoff-inspired dominance relation for preference-based evolutionary multi-objective optimization," *Swarm Evol. Comput.*, vol. 63, p. 100866, Jun. 2021, doi: 10.1016/J.SWEVO.2021.100866.

Knowledge Retrieval Architectures for AI-Assisted Software Development: From Static Context to Autonomous Development Agents

Norbert Fijałek [0009-0004-2756-0451],
Ilona Bluemke [0000-0002-2894-5976],
Piotr Gawrysiak [0000-0002-9647-6761]

Faculty of Electronics and Information Technology
Warsaw University of Technology
Warsaw, Poland
norbert.fijalek.dokt@pw.edu.pl

Abstract-**The integration of Large Language Models (LLMs) into software development workflows has fundamentally transformed how developers interact with knowledge repositories, codebases, and documentation. However, traditional knowledge retrieval mechanisms face significant challenges when applied to the dynamic, multi-modal, and contextually-rich environment of software engineering. This survey provides a comprehensive analysis of knowledge retrieval architectures specifically designed for AI-assisted software development, examining the evolution from static context provision to autonomous development agents. Building upon this systematic examination, the paper delineates critical research directions that advance the theoretical and practical foundations of AI-assisted software development.**

Keywords-Knowledge Retrieval, Large Language Models, Retrieval-Augmented Generation, Agentic RAG, Graph RAG, AI-Assisted Software Development, AI SDLC.

I. INTRODUCTION

The rapid advancement of Large Language Models has created opportunities for AI-assisted software development, yet their effectiveness fundamentally depends on how they access and utilize knowledge. While LLMs demonstrate remarkable capabilities in code generation, their reliance on static parametric knowledge creates limitations in software engineering's dynamic environment, where frameworks and libraries evolve continuously. This challenge intensifies when enterprise codebases contain millions of lines of code with complex interdependencies that exceed even advanced models' context windows. This survey addresses a critical gap in existing literature: while numerous studies examine individual retrieval architectures or specific AI coding assistants, no comprehensive analysis systematically compares knowledge retrieval approaches specifically designed for software development contexts. This work provides a unified framework for understanding how different architectural approaches handle the distinct challenges of software development knowledge retrieval. This survey employs a systematic literature review methodology inspired by the guidelines proposed by Kitchenham [30], adapted to the specific characteristics of knowledge retrieval research in AI-assisted software development.

The review was guided by three research questions: (RQ1) What knowledge retrieval architectures have been proposed or applied in AI-assisted software development, and what are their core mechanisms? (RQ2) What challenges do these architectures face when applied to software engineering contexts? (RQ3) What gaps and future directions emerge from the current body of work? The search was conducted across three databases: IEEE Xplore, ACM Digital Library, and arXiv, covering publications from 2020 to 2025 to capture the evolution following transformer-based models' widespread adoption. The search strategy employed combinations of the following terms: "retrieval-augmented generation" AND "software development"; "knowledge retrieval" AND "code generation"; "graph RAG" OR "agentic RAG"; "LLM" AND "software engineering" AND "retrieval"; and "AI coding assistant" AND "knowledge". Additionally, supplementary sources were identified through citation tracking of key foundational works-including seminal contributions on few-shot learning capabilities of large language models [5], prompt engineering patterns and programming paradigms [3, 4], chain-of-thought reasoning [6], and the foundational retrieval-augmented generation framework [7]-as well as through targeted searches in Google and Google Scholar for works not indexed in the primary databases. Subsequent survey literature examining RAG system design and limitations [8] further informed the scope of the review. Two foundational Polish-language textbooks on artificial intelligence for engineers [1-2] were included as they provide comprehensive theoretical frameworks for AI paradigms.

The initial search yielded approximately 100 candidate sources. Inclusion criteria required that works: (1) address knowledge retrieval mechanisms in the context of AI-assisted software development or closely related code generation tasks; (2) be published in peer-reviewed venues, established preprint repositories, or constitute recognized reference works; (3) be written in English or Polish. Exclusion criteria removed works that: (1) focused exclusively on general-purpose RAG without software engineering application; (2) were short workshop abstracts or position papers lacking technical depth; (3) were duplicates or earlier versions superseded by more comprehensive publications; (4) lacked methodological substance. After applying these criteria, 50 candidate sources remained for full-text review. From these, a final set of 30

Manuscript received June 22, 2026; revised July 08, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.147

works was selected based on their direct relevance to knowledge retrieval architectures for software development, methodological rigor, and contributions to system design principles, prioritizing works addressing software engineering's unique challenges. The classification of architectures into four paradigms (Context Window and Prompting, Traditional RAG, Graph RAG, and Agentic RAG) emerged from iterative analysis of the extracted data, with each work assigned to one or more categories based on its primary contribution, where section II establishes foundational concepts of LLM knowledge handling and retrieval-augmented approaches. Section III analyzes five interconnected challenge categories: information veracity, contextual understanding, scale and performance, workflow integration, and knowledge representation. Section IV surveys available architectures-Context Window and Prompting, traditional RAG, Graph RAG, and Agentic RAG-examining their mechanisms, strengths, and limitations. Section V presents future directions of research in terms of hybrid agentic architectures.

II. FUNDAMENTALS

Richard Hamming's words that "the purpose of computing is insight, not numbers" [20] has never been more relevant than in the current era of Large Language Models, which have fundamentally transformed how artificial intelligence systems store and access knowledge. Traditional models keep all their information in parameters learned during training, which works well for general language tasks but creates significant problems in specialized fields like software development [5]. The challenge becomes particularly apparent in software engineering, where programming languages, frameworks, and libraries evolve much faster than any training process can capture.

This limitation of static knowledge has led to the development of RAG (Retrieval-Augmented Generation), which represent a major advancement in AI systems [7]. The necessity for such approaches becomes evident when considering that classical computer science methods, where deterministic algorithms are constructed to solve problems, have in many cases reached a development barrier [1]. While traditional approaches rely solely on pre-trained knowledge, retrieval-augmented systems can access external information sources to stay current with rapidly changing technical domains. The knowledge encoded during training becomes like a snapshot frozen in time, making it impossible to incorporate new developments that are essential for effective software development support.

The foundation of modern retrieval systems lies in vector space models and sophisticated knowledge representation techniques. These systems create high-dimensional spaces where related concepts cluster together, allowing computers to calculate similarity using methods such as cosine similarity. Knowledge representation involves symbolic encoding of facts that agents recognize as true, with mapping between facts and their representations in given systems [2]. This approach helps identify relevant code examples based on their meaning rather than just matching keywords. Current code embeddings tend to capture surface-level similarities more effectively than deeper semantic relationships [8]. This creates a particular challenge when working with different types of information-code, documentation, and architectural diagrams all require different approaches to knowledge extraction and representation. Context window limitations present another significant obstacle for software engineering applications. Modern software projects often require understanding context that far exceeds what current language models can handle [21]. Even with recent advances like Gemini 1.5, which can process millions of tokens [10], the technology still falls short of enterprise-scale requirements.

The evolution from static to dynamic AI systems reflects broader patterns in artificial intelligence development. The field of natural language processing has evolved from classical rule-based approaches to statistical and deep learning methods, with historical approaches using formal language knowledge while modern approaches rely on statistical analysis of large text datasets and neural networks trained on massive corpora [2]. Modern IDEs (Integrated Development Environment) improved with semantic analysis understanding variable scope and type systems, but remained fundamentally reactive without broader architectural understanding. AI-powered coding assistants mark a major shift toward sophisticated knowledge retrieval systems that address the mechanization and engineering instrumentation challenges inherent in creating methods for solving technical problems [1]. These systems must handle the full complexity of modern software development: code generation, transformation, testing, analysis, and configuration management [18]. Each area requires different strategies, showing that single-approach solutions aren't adequate. This diversity raises questions about whether there exists an optimal architecture for all use cases, or whether specialized architectures would be more appropriate.

Recent studies have documented these limitations in real-world scenarios. Research demonstrates how AI coding assistants encounter difficulties with enterprise-scale codebases that exceed current context windows by several orders of magnitude [17]. Similarly, studies found that effectiveness varies significantly based on programming language coverage in training data and the amount of contextual information that can be accommodated within model constraints [16]. These findings raise important questions about whether specialized training approaches might be needed for software development contexts.

Modern retrieval systems must also handle the multi-modal nature of software engineering knowledge. Effective development assistance requires combining textual documentation, code examples, architectural diagrams, API specifications, and configuration files. Traditional text-based retrieval methods prove insufficient for this diverse information environments and need specialized architectures that can navigate complex knowledge landscapes.

These foundational concepts reveal both the promise and limitations of current knowledge retrieval approaches in software engineering. The subsequent sections examine the

practical challenges these systems face (Section III) and how different architectural paradigms address these constraints (Section IV).

III. Unified Challenges in Knowledge Retrieval for AI-Assisted Software Development

Building AI systems that can effectively help software developers faces several interconnected challenges. These difficulties arise from both the limitations of current AI technology and the specific nature of software development work. Importantly, the complexity and severity of these challenges differ significantly between two levels of AI assistance [17]: basic coding assistants (focused primarily on code completion and syntax suggestions) and comprehensive software engineering support (requiring architectural understanding, system design decisions, and cross-component reasoning) [18]. Understanding these problems is important because they affect each other and cannot be solved separately, and their manifestations vary dramatically depending on whether the AI aims to assist with isolated coding tasks or complete software engineering workflows. This section systematically examines five interconnected challenge categories that emerge from the literature. We begin with Information Veracity and Reliability Challenges, exploring how AI systems struggle with accuracy and explainability in code generation. Next, Contextual Understanding and Integration Challenges addresses the complexities of combining multi-modal knowledge across different technical domains. Scale and Performance Challenges examines computational limitations and sustainability concerns when processing enterprise-scale codebases. Development Workflow Integration Challenges investigates the practical barriers to incorporating AI systems into existing development ecosystems. Finally, Knowledge Representation and Modeling Challenges analyzes the unique characteristics of software engineering knowledge that distinguish it from other AI application domains. Each subsection identifies key works from our systematic literature review that document these challenges, providing a comprehensive foundation for understanding why current retrieval architectures face fundamental limitations (examined in Section IV) and why hybrid approaches become necessary (explored in Section V).

A. Information Veracity and Reliability Challenges

Software development requires precise information because wrong answers can lead to serious problems like system crashes or security breaches. Recent studies of AI coding assistants in production environments reveal consistent patterns of accuracy challenges, particularly in code generation hallucinations [8] and performance degradation at enterprise scale [17]. AI systems make mistakes in different ways when helping developers. Sometimes they create code that looks correct but breaks programming rules or uses features that don't exist [8]. This happens more often with less popular programming languages because the AI systems have less training data about them, causing them to incorrectly apply patterns from other well-known languages like Python or JavaScript [18].

Another common problem occurs when AI systems generate code that follows grammar rules but doesn't work in the real context. For example, they might suggest using a function that doesn't exist or use an existing function in the wrong way [16]. Real-world studies show that AI-generated code often contains subtle errors that developers need to catch and fix manually [17]. Making matters worse, software technologies change constantly. New versions of programming languages and frameworks regularly introduce changes that can make existing knowledge outdated within just a few months [18].

A fundamental challenge lies in the explainability of AI-generated solutions. Often, problems are caused by the so-called explainability issue-demonstrating why the result of the AI mechanism for a given set of data is this and not another [1]. This lack of transparency becomes particularly problematic in software development contexts where developers need to understand not just what code to write, but why specific approaches are recommended. Responsible AI principles emphasize ensuring that AI system outputs are beneficial and non-intrusive for users, especially when dealing with large populations [2]. Systems must be designed to prevent the perpetuation of existing inequalities or discriminatory decision-making. Ensuring data diversity and representativeness used for training AI systems is crucial for limiting bias [2]. In software development contexts, this translates to ensuring that AI coding assistants don't perpetuate poor coding practices or introduce systematic biases that could affect software quality or security.

B. Contextual Understanding and Integration Challenges

Modern software development requires combining knowledge from many different technical areas. A single programming task might need understanding of how code works, specific framework patterns, database design, security requirements, and performance optimization [7]. Research examining multi-modal knowledge processing and enterprise codebase navigation [2, 7, 10] reveals that the challenge isn't just finding information about each area-it's combining this knowledge while avoiding conflicts between different approaches. The complexity of contextual understanding is further complicated by the inherent characteristics of natural language processing in software contexts. Natural language is characterized by changing vocabulary and semantic functions, fluidity and incompleteness of syntax rules [2]. This means that natural language is neither a closed nor finite system. The semantics of sentences can change over time, and it's never possible to definitively determine whether a sentence belongs to the language [2]. When AI systems must interpret developer queries, documentation, and code comments, these linguistic challenges compound the technical complexity of providing accurate assistance.

Additionally, software development happens in constantly changing environments. Code repositories grow and evolve every day, and software dependencies get updated regularly. This creates complex relationships between different parts of a system that are rarely documented properly [10]. Current AI systems struggle to understand these complex ecosystems

where multiple code repositories and services depend on each other in complicated ways.

C. Scale and Performance Challenges

Today's software projects are often much larger than what current AI models can handle. Large enterprise applications can contain millions of lines of code spread across thousands of files with complex dependencies [9]. Studies of large codebase processing and real-time response optimization [9-11] demonstrate that even the most advanced AI models like Sonnet 3.5 [28], which can process millions of tokens, still face practical limitations when dealing with real-world enterprise software [10].

The computational demands of modern AI systems present significant sustainability challenges. Training large language models requires energy comparable to what's needed for flights between Pacific and Atlantic coasts of the USA. Energy consumption for larger models like GPT-3 or GPT-4 is comparable to that needed to light a small city [2]. This creates a critical tension between the desire for more capable AI coding assistants and the environmental impact of the computational resources required to train and operate them.

Speed remains a critical issue. Developers expect quick responses from their tools. If an AI assistant takes more than a few seconds to respond, it significantly hurts productivity. This creates pressure to optimize systems in ways that might reduce the quality of answers [11]. Designing new AI methods to ensure high-quality operation while maintaining appropriate energy efficiency is essential [2] for creating sustainable and practical development assistance tools.

D. Development Workflow Integration Challenges

Integrating AI systems into existing development processes involves more than just technical challenges. Modern software development relies on complex tool ecosystems including code editors, version control systems, and automated testing and deployment systems [12]. Research on tool ecosystem compatibility, security concerns, and developer adoption patterns [12, 14, 17] shows that AI systems must work smoothly with all these tools while maintaining compatibility across different technology stacks, yet technical capabilities alone cannot ensure successful deployment. The sensitive nature of software development information creates significant privacy and security concerns. Code repositories often contain proprietary business logic and confidential information that needs protection [14]. Building trust among developers presents another barrier, with concerns about job security and becoming too dependent on automated systems [17].

E. Knowledge Representation and Modeling Challenges

Software engineering knowledge has unique characteristics that make it different from other areas where AI is used. Code serves both as instructions for computers and as communication between developers, creating complexities that traditional text-based approaches cannot handle properly [5]. Analysis of code semantics, multi-level dependency structures, and multi-modal information integration [3, 5-6, 13] demonstrates that current analysis methods focus mainly on grammar and structure, but effective assistance requires understanding deeper relationships that go beyond correct syntax [3]. Software systems show complex relationship patterns that exist at multiple levels. These include direct connections between modules, indirect coupling through shared data structures, and semantic relationships based on similar functionality [13]. Software knowledge also exists in multiple formats: documentation, code, diagrams, and configuration files. Effective systems need approaches that maintain consistency while preserving the unique characteristics of each type of information [6]. These challenges in AI-assisted software development form a connected web where problems with information accuracy, context understanding, scale limitations, workflow integration, and knowledge representation make each other worse [18, 11]. These fundamental limitations require specialized architectural approaches that can handle software engineering's unique characteristics: multiple types of knowledge, constantly changing information, and complex relationships across different levels of abstraction [10, 12].

IV. Overview of Available Architectures

The systematic literature review reveals a clear evolutionary trajectory in knowledge retrieval architectures for AI-assisted software development, progressing from simple static approaches toward increasingly sophisticated autonomous systems. Analysis of selected works demonstrates that researchers and practitioners have consistently sought to address the fundamental tension between system capability and implementation complexity. The literature shows four distinct architectural paradigms that have emerged in response to software engineering's unique challenges: Context Window & Prompting, which dominated early LLM applications; Retrieval-Augmented Generation (RAG), which gained prominence following the 2021 introduction of the RAG framework [7]; Graph RAG, which emerged in 2024 as researchers recognized the limitations of vector-based approaches for capturing complex software dependencies [9]; and Agentic RAG, which represents the current frontier with multi-agent systems appearing predominantly in 2023-2024 literature [12-14]. Each architectural paradigm addresses specific limitations of its predecessors while introducing new complexities, creating what the literature characterizes as a fundamental trade-off between sophisticated capabilities and practical deployment feasibility. This section examines each architecture systematically, analyzing how they handle the five challenge categories identified in Section III, and explaining why no single approach currently provides a complete solution for all software development contexts.

A. Context Window and Prompting

The simplest approach involves putting all necessary information directly into the AI model's input window. This method doesn't retrieve external knowledge but instead relies on carefully written prompts that include relevant context alongside the user's question [5]. This approach aligns with the symbolic artificial intelligence paradigm, which was dominant in early AI development-GOFAI (Good Old-fashioned Artificial Intelligence) [24] and uses high-level, human-understandable representation of problems [1]. Smart

prompting has become quite sophisticated over the years. Developers can create templates that work well with different programming languages by including language-specific patterns and common frameworks in their prompts [3]. These templates help maintain consistency across different coding tasks, whether generating new code, debugging problems, or writing documentation. Chain-of-thought prompting represents another advancement, where the AI breaks down complex problems into smaller steps, making the reasoning process more transparent [6]. Recent research has demonstrated that effective prompt engineering can significantly improve code generation quality and reduce the need for post-generation corrections [4].

The key strength of this approach lies in its simplicity and speed. When developers need quick answers to straightforward questions, context window prompting delivers immediate results. The technique works particularly well with few-shot learning, where providing a few examples helps the AI understand patterns and apply them to new situations [5]. However, the limitations become obvious when working with large codebases. As noted earlier, modern enterprise software projects contain millions of lines of code spread across thousands of files, far exceeding what any current AI model can process at once [10]. This forces developers to carefully select which information to include, often missing important context that could affect the AI's recommendations. Studies have shown that context window limitations significantly impact the quality of AI assistance in complex software engineering tasks [18].

B. Retrieval-Augmented Generation (RAG)

RAG systems take a different approach by first searching through external knowledge sources before generating responses. When a developer asks a question, the system finds relevant information from documentation, code repositories, or other knowledge bases, then provides this context to the AI model along with the original question [7]. This approach leverages statistical artificial intelligence methods, utilizing mathematical tools that allow for solving problems such as vector similarity and retrieve relevant information [1].

Building effective RAG systems for software development requires understanding how code differs from regular text. Code serves both as instructions for computers and communication between developers, creating unique challenges for knowledge retrieval [8]. These systems need specialized embedding models that can capture programming concepts and multiple indexing strategies to handle different types of relationships in software projects. The integration of external knowledge sources has been shown to significantly improve the accuracy and relevance of AI-generated code compared to purely parametric approaches [7]. Modern RAG systems employ reasoning engines that derive new facts from existing explicitly represented facts and axioms, enabling them to make intelligent connections between retrieved information [2].

One critical decision involves how to break down large codebases into manageable pieces. Function-level chunking treats each function as a separate unit, which works well for specific implementation questions but struggles with broader architectural concerns. File-level chunking keeps entire classes and modules together, preserving more design context but potentially including irrelevant information. Repository-level chunking captures the most context but risks overwhelming the system with too much information. RAG systems show significant improvements over basic prompting for many software development tasks. They can access current documentation, find relevant code examples from large repositories, and stay updated with recent changes. However, they still operate reactively, responding to specific queries rather than proactively understanding the developer's broader goals. Empirical evaluations have demonstrated that RAG-based systems consistently outperform traditional prompting approaches in code completion and documentation generation tasks [11]. Recent evaluation frameworks (such as RAGAS) [25] have established standard ways to measure RAG system performance across different tasks like fact verification and reasoning capabilities [11]. These metrics help organizations choose appropriate systems for their specific needs and optimize performance over time.

C. Graph RAG

While traditional RAG systems treat knowledge as composed of independent pieces of information, Graph RAG recognizes that software development involves complex relationships between different components. This approach builds on semantic networks, which provide graphical notation for knowledge represented as sets of nodes (concepts) connected by labeled arcs describing relationships between nodes [2]. Knowledge graphs serve as large, graph-structured knowledge bases that represent facts as relationships between entities, with basic components including entities expressed as graph nodes, their properties (attributes), and relations connecting nodes expressed as graph edges [2]. Creating these graphs requires sophisticated algorithms that can identify software engineering entities and understand how they relate to each other. Direct dependencies between modules represent the most obvious relationships, but software systems also have semantic connections based on shared functionality and temporal associations that track how code evolves over time. Entities in these knowledge graphs can have semantic types represented through is-a relationships between entities and their types, enabling more sophisticated understanding of code architecture [2]. Research has shown that graph-based representations can capture complex software dependencies more effectively than traditional flat document structures [9]. Graph RAG systems excel in scenarios that require understanding of these relationships. When developers need to assess the impact of proposed changes, these systems can trace dependency paths across multiple levels of software architecture. They help identify potential breaking changes, security implications, and performance considerations that might not be obvious from looking at individual code pieces. The contemporary manifestation of semantic networks through technologies like RDF, RDFS, and OWL enables the creation and dissemination of semantic networks and ontologies in standard form, facilitating knowledge sharing across

development teams [2]. The ability to traverse graph structures also enables comprehensive architectural understanding. Developers can explore system architecture from multiple perspectives, understanding not just what code does but how different parts interact and depend on each other. Query-focused summarization capabilities allow these systems to generate summaries that capture complex relationships and dependencies that simpler vector-based approaches would miss [9].

Despite these advantages, Graph RAG systems face significant challenges in maintenance and scalability. Building accurate graphs requires substantial computational resources, and keeping them updated as codebases evolve presents ongoing technical challenges. Studies indicate that the overhead of maintaining graph structures can be prohibitive for rapidly changing software projects [14]. These maintenance challenges echo earlier Semantic Web initiatives, where ontology evolution and curation consistently emerged as primary barriers to adoption [29]. Manual construction and maintenance of formal knowledge structures proved unsustainably labor-intensive, particularly in dynamic domains-challenges that software engineering's rapid technological evolution amplifies further.

D. Agentic RAG

The most advanced approach integrates autonomous agents with knowledge retrieval, creating systems that can reason independently and handle complex, multi-step tasks [12]. This approach exemplifies the subsymbolic artificial intelligence paradigm, which unlike symbolic and statistical approaches, does not assume any specific form of knowledge representation, allowing for more flexible and adaptive reasoning capabilities [1]. Unlike previous approaches that simply respond to developer questions, agentic systems maintain persistent memory, break down complex objectives into smaller tasks, and coordinate multiple operations over extended periods.

These systems represent a fundamental shift in how AI assists with software development. Rather than functioning as advanced search engines, they operate more like collaborative team members who can understand project goals, plan implementation strategies, and execute comprehensive workflows [12]. Multi-agent frameworks enable different specialized agents to work together, each contributing specific expertise while coordinating toward common objectives. Research has demonstrated that multi-agent systems can handle complex software engineering tasks that exceed the capabilities of single-agent approaches [13].

Agentic RAG systems can integrate directly with development tools, version control systems, testing frameworks, and deployment pipelines. This integration transforms them from passive information providers into active participants in the development process. They can autonomously execute testing procedures, manage code deployments, and even monitor system performance. Advanced implementations include learning mechanisms that enable continuous improvement through experience and feedback [13]. These systems remember successful strategies, learn from common failure patterns, and develop more effective approaches over time. The reasoning capabilities inherent in these systems allow them to derive new facts and insights from existing knowledge, enabling them to make intelligent decisions about development workflows [2]. The rapid progress in multi-agent systems research demonstrates significant potential while highlighting ongoing challenges in coordination, communication, and scalability [14].

The emergence of these sophisticated systems has already begun changing how developers work. Organizations report that AI systems increasingly function as team members rather than tools, creating new collaboration patterns that represent a significant shift in professional software development [17].

E. Architectural Evolution and Analysis

The progression from simple prompting to agentic systems shows clear advancement in both knowledge representation and autonomous capabilities. Each approach builds on previous paradigms while introducing new ways to handle software engineering's complex knowledge requirements. This evolution reflects the broader development of artificial intelligence approaches, from symbolic through statistical to subsymbolic paradigms, each offering distinct advantages for different aspects of software development assistance [1]. To systematically compare the characteristics of the identified architectural paradigms, Table I summarizes their key properties across four evaluation dimensions: knowledge access method, setup complexity, hallucination risk, and autonomy level. This comparison enables a structured assessment of the trade-offs between capability sophistication and practical deployment feasibility.

TABLE I. A COMPARISON ACROSS KEY DIMENSIONS OF DIFFERENT KNOWLEDGE RETRIEVAL ARCHITECTURES.

Architecture	Knowledge Access	Setup Complexity	Hallucination Risk	Autonomy Level
Context Window & Prompting	Manual/Static	Low	Medium	Low
Traditional RAG	Automated/Limited	Medium	Medium	Low
Graph RAG	Relational/Structured	High	Low-Medium	Medium
Agentic RAG	Multi-source/Dynamic	High	Low	High

This comparison highlights a fundamental pattern: more sophisticated architectures offer superior capabilities but require significantly more complex implementation and maintenance. Simple prompting and traditional RAG provide immediate practical value with straightforward deployment, while Graph RAG and Agentic RAG deliver advanced knowledge representation and autonomous capabilities at much higher implementation costs. Recent evaluation frameworks (such as RAGAS) [25] provide systematic approaches for comparing these architectures across multiple dimensions, helping organizations make evidence-based decisions about which approaches best fit their specific software development contexts [11]. While each architectural paradigm offers distinct

advantages for particular scenarios, none completely addresses all the complex requirements of modern software development environments. The limitations of individual approaches point toward the need for hybrid systems that combine strengths from multiple paradigms while maintaining practical deployment feasibility.

V. New Directions-Hybrid Agentic Architectures

Current knowledge retrieval systems for software development work well in some situations but fall short when developers need comprehensive assistance with complex projects. Each architecture examined in Section IV has clear strengths, yet none can handle all the challenges that modern software engineering presents. This gap points toward the need for hybrid approaches that combine different retrieval methods within intelligent agent systems-an area that remains largely unexplored and represents a critical direction for future research.

A. Theoretical Foundations for Hybrid Systems

Traditional RAG systems struggle to maintain context when working with large codebases that span multiple files and complex relationships [8]. Future hybrid agentic approaches could potentially address this limitation by employing autonomous agents that intelligently select which retrieval strategies matter most for any given task. These agents would need to maintain context across extended development sessions-a capability current systems lack. Developing such context-aware orchestration represents a fundamental research challenge requiring advances in multi-strategy coordination and adaptive retrieval.

B. Unified Knowledge Representation

A promising research direction involves investigating unified data processing approaches that convert all software engineering artifacts-code files, documentation, diagrams, configuration files, and version control history-into standardized formats before retrieval processing begins. This addresses a fundamental principle in artificial intelligence: representation matters [1]. Just as the multiplication of numbers is trivial in Arabic numerals but nearly impossible in Roman numerals, the effectiveness of knowledge processing depends critically on how information is represented [1]. By establishing common semantic foundations, future systems could enable both vector-based embeddings and graph-based relationship extraction to work with consistent data structures. The multi-modal nature of software engineering knowledge presents unique challenges that hybrid architectures would be well-positioned to address [2]. Modern software development involves processing diverse information types simultaneously-code, documentation, architectural diagrams, and configuration files-each requiring specialized processing methods yet ultimately working together to provide comprehensive development assistance [2]. This would allow seamless integration of insights from multiple information types when responding to developer queries-something current systems cannot achieve due to format inconsistencies across different retrieval methods.

C. Multi-Engine Integration Strategies

Future hybrid architectures should explore integration of multiple complementary engines: vector-based systems for semantic similarity, graph-based systems for relationship analysis, and agent-based systems for intelligent orchestration. The central research challenge lies in developing effective coordination mechanisms that can determine which retrieval strategy best fits specific query types, merge results from different retrieval methods intelligently, resolve conflicts, and assess confidence levels across heterogeneous information sources [11]. Such systems would need to run multiple retrieval operations in parallel while synthesizing results through sophisticated decision-making algorithms [12]-capabilities that require significant algorithmic advances beyond current state-of-the-art.

D. Agent-Based Knowledge Management

Integration of agent-based systems could draw from established principles in robotics and knowledge representation [2]. Frame-based knowledge representation might help autonomous agents understand their environment and interact effectively with software development contexts [2]. These agents would need to continuously maintain relationship mappings while keeping embeddings updated, both working from unified source representations. However, developing efficient parallel processing strategies and adaptive learning mechanisms represents substantial research challenges. Agents must be capable of managing knowledge base evolution independently-incorporating new information sources, updating existing structures, and retiring outdated information while maintaining semantic coherence [13].

E. Critical Evaluation Dimensions

Evaluating future hybrid agentic architectures will require assessing performance across multiple critical dimensions. These include strategy selection effectiveness-how well agents choose optimal retrieval method combinations; cross-modal integration quality-how successfully agents synthesize results from vector and graph-based approaches; learning and adaptation-how quickly agents improve strategy selection through experience; resource optimization-how efficiently agents balance computational costs across different retrieval modes; and temporal awareness-how effectively systems track knowledge evolution and maintain currency.

VI. Future Work

This survey points to several important research directions that require further investigation. A key area involves developing algorithms that can select the best retrieval strategy in real time, combined with techniques for merging information from different sources while keeping results consistent [11]. Equally important is finding ways to track rapidly changing software knowledge and making retrieval and reasoning processes more transparent and explainable [1]. Knowledge representation must move beyond current embeddings and graph structures to capture how software evolves over time-including changing requirements, code transformation patterns, and technical decision cascades. Successful deployment

requires not only technical progress but also organizational adaptation [17], while the regulatory landscape-particularly GDPR [26] and the EU AI Act [27]-has made explainable AI a critical requirement. These directions converge toward building genuinely collaborative systems that work alongside human developers rather than replacing them [17], maintaining transparency and meaningful human control while serving the broader goal of creating better software that benefits society.

VII. Conclusions

The evolution from static documentation to autonomous coding agents mirrors a broader truth about technological progress: each generation of tools doesn't simply do more-it fundamentally reshapes what becomes possible, transforming yesterday's architectural visions into tomorrow's reality. This survey systematically provides a unified theoretical and empirical foundation for understanding knowledge retrieval architectures in AI-assisted software development. The principal contributions are threefold: (1) a unified taxonomy of challenges across five interconnected dimensions demonstrating how limitations in one dimension exacerbate others; (2) a systematic comparative analysis of four architectural paradigms revealing a fundamental trade-off between capability and implementation complexity; (3) the identification of critical research directions toward hybrid agentic architectures requiring algorithmic innovations beyond the current state-of-the-art. The analysis reveals that retrieval architectures have progressed through distinct evolutionary stages-from symbolic paradigms through statistical approaches to subsymbolic methods-mirroring broader AI development patterns [1], with no single approach adequately addressing all modern software development requirements. Current systems demonstrate measurable improvements in code quality and developer productivity [16], yet struggle with complex architectural reasoning that experienced developers perform intuitively [15, 17]. This gap points toward hybrid architectures combining multiple retrieval mechanisms within autonomous agent frameworks. Research on memory-augmented systems and continual learning [19]-showing that effective retrieval must span factual memory, sense-making, and associative reasoning-provides evidence that such approaches are theoretically sound, though significant technical barriers remain.

As software systems continue to grow in scale and complexity, advancing knowledge retrieval architectures from static context provision to genuinely autonomous development agents remains both a promising and necessary direction for research and practice.

References

[1] M. Muraszkiewicz and R. Nowak (eds.), Sztuczna inteligencja dla inżynierów - metody ogólne. Warsaw, 2022 (in Polish).

[2] M. Muraszkiewicz and R. Nowak (eds.), Sztuczna inteligencja dla inżynierów - istotne obszary i zastosowania. Warsaw, 2023 (in Polish).

[3] J. White et al., "A prompt pattern catalog to enhance prompt engineering with ChatGPT," in PLoP '23, Article 5, pp. 1–31, 2023.

[4] L. Reynolds and K. McDonell, "Prompt programming for large language models: beyond the few-shot paradigm," in CHI EA '21, Article 314, pp. 1–7, 2021.

[5] T. Brown et al., "Language models are few-shot learners," in NIPS'20, Article 159, pp. 1877–1901, 2020.

[6] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in NIPS'22, Article 1800, pp. 24824–24837, 2022.

[7] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in NIPS'20, Article 793, pp. 9459–9474, 2020.

[8] Y. Gao et al., "Retrieval-augmented generation for large language models: a survey," arXiv:2312.10997, 2023.

[9] D. Edge et al., "From local to global: a Graph RAG approach to query-focused summarization," arXiv:2404.16130, 2024.

[10] Google Gemini Team, "Gemini 1.5: unlocking multimodal understanding across millions of tokens of context," arXiv:2403.05530, 2024.

[11] S. Krishna et al., "Fact, fetch, and reason: a unified evaluation of retrieval-augmented generation," in Proc. NAACL-HLT, vol. 1, 2025.

[12] Q. Wu et al., "AutoGen: enabling next-gen LLM applications via multi-agent conversation," in COLM, Philadelphia, 2024.

[13] T. Guo et al., "Large language model based multi-agents: a survey of progress and challenges," in IJCAI '24, Article 890, pp. 8048–8057, 2024.

[14] Z. Xi et al., "The rise and potential of large language model based agents: a survey," Sci. China Inf. Sci., vol. 68, 121101, 2025.

[15] A. M. Dakhel et al., "GitHub Copilot AI pair programmer: asset or liability?" J. Syst. Softw., vol. 203, 2023.

[16] V. Viswanadhapalli, "AI-augmented software development: enhancing code quality and developer productivity using large language models," IJNRD, vol. 9, no. 8, pp. e382–e396, 2024.

[17] N. S. Shashidhara, "AI in software engineering - how intelligent systems are changing the software development process," EJCSIT, vol. 13, no. 29, pp. 28–39, 2025.

[18] A. Gu et al., "Challenges and paths towards AI for software engineering," arXiv:2503.22625, 2025.

[19] B. Gutierrez et al., "From RAG to memory: non-parametric continual learning for large language model," in ICML, PMLR 267, pp. 21497–21515, 2025.

[20] R. W. Hamming, Introduction to Applied Numerical Analysis. New York: Dover, 1989.

[21] Interesting Engineering, "What's the biggest software package by lines of code?" 2021. [Online]. Available: https://interestingengineering.com/lists/whats-the-biggest-software-package-by-lines-of-code

[22] FAISS Documentation. [Online]. Available: https://faiss.ai/index.html

[23] Neo4j Documentation. [Online]. Available: https://neo4j.com/docs/

[24] The Cambridge Handbook of Artificial Intelligence. Cambridge: Cambridge University Press, 2014.

[25] RAGAS Documentation. [Online]. Available: https://docs.ragas.io/en/stable/

[26] European Parliament, "The impact of the General Data Protection Regulation (GDPR) on artificial intelligence," 2020.

[27] Regulation (EU) 2024/1689 (Artificial Intelligence Act), 2024. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2024/1689/oj/eng

[28] E. Alor et al., "Evaluating the use of LLMs for documentation to code traceability," arXiv:2506.16440, 2025.

[29] G. Flouris et al., "Ontology change: classification and survey," The Knowledge Engineering Review, vol. 23, no. 2, Cambridge University Press, 2008.

[30] B. Kitchenham, "Procedures for performing systematic reviews," Joint Technical Report, Software Engineering Group, Dept. of Computer Science, Keele University, 2004.

SIAM DSL: Accessible Structured Text for Bridging the Architecture-Impact Gap in Cyber-Physical Production Systems

Stefan Biffl*x Tobias Bein† Sebastian Kropatschekx Kristof Meixner* Arndt Lüder†
*Institute of Information Systems Engineering, TU Wien, Vienna, Austria
xCenter for Digital Production, TU Wien, Vienna, Austria
E-Mail: [first].[last]@tuwien.ac.at
†Institute of Engineering of Products and Systems, Otto von Guericke University, Magdeburg, Germany
E-Mail: [first].[last]@ovge.de

***Abstract*—In semi-automated Cyber-Physical Production Systems (CPPSs) that use operator assistance systems, introducing product variants often induces severe hazards due to misaligned mental models between the assistance system, human operators, and actual production states. These discrepancies create high-impact deviations between the expected and actual runtime states of a production line. To prevent these socio-technical hazards, solution architects require early-stage elicitation of tacit domain knowledge regarding critical production paths and operator conditions. However, multi-domain modeling approaches remain largely inaccessible to production domain experts. To bridge this gap, we present the *System Impact Architecture Model (SIAM) Domain-Specific Language (DSL)*, a lightweight, structured text language that integrates a System Architecture Model with a System Impact Model. The SIAM DSL provides a simple framework for capturing and comparing multi-domain Product-Process-Resource (PPR) variations, facilitating forward and backward reasoning to identify risky production states. We explore the language's efficacy through an illustrative case study of operator assistance design for gearbox assembly and derive a research agenda. The study results indicate that the SIAM DSL is effective for eliciting tacit dependencies and providing input to modeling socio-technical assistance scenarios.**

***Index Terms*—Production Systems Engineering; Industry 4.0; Domain-Specific Language; Model-Based Engineering**

I. Introduction

Industrial digitalization has become an established reality across manufacturing sectors, driving expectations for enhanced system performance and operational outcomes [1]. While Industry 4.0 (I4.0) has focused on increasing flexibility through highly automated Cyber-Physical Production Systems (CPPSs) [2], the emerging Industry 5.0 vision emphasizes socio-technical production systems that integrate automation with human operators to address changes in a volatile environment [3]. In these systems, Operator Assistance Systems (OASs) are critical for providing real-time guidance and observing task execution to improve the performance of human-machine teams [4]. Despite the potential of OASs, introducing product variants frequently induces severe production hazards and low performance [4], [5].

According to *STAMP* theory [6], a socio-technical production system can be seen as a control system. Production hazards emerge from dysfunctional control systems in which the mental models of various actors, e.g., operators, CPPS engineers, and OAS designers, diverge from the actual system state, leading to unsafe actions. For instance, an operator fails to set the correct pressing force, resulting in invisible defects such as hair cracks, which reduce business performance.

These divergent model views stem from *architecture-impact gaps* among domain experts. In this paper, we define an *architecture-impact gap* as the systemic difference between design-time engineering models of the intended architecture and the actual run-time states of the production line, the operational impact. In multi-domain production engineering, this gap may arise from the use of a collection of partial architecture views, models, and Domain-Specific Languages (DSLs). These have traditionally focused on a fixed Product-Process-Resource (PPR) set for production [7], making it hard to model production states that contribute to emerging hazards due to flexible products, processes, or resources. Following Leveson [6], these diverging model views can be interpreted as open control loops that need to be closed by system design [8].

Bridging architecture-impact gaps is complicated, as the engineering and operation of CPPSs encompass diverse knowledge-intensive processes [9]. Critical domain knowledge on PPR assets is often tacit and scattered across disciplines [10], including product design, mechanical engineering, and quality management, each with distinct engineering habits that are hard to integrate [11]. Unfortunately, traditional modeling frameworks, such as RAMI 4.0 [12], are effective for representing static structures but lack the causal logic required for impact prediction. Graphical DSLs are often too complex for non-modeling experts [13], considering production variants, while simple structured-text languages like *Gherkin* [14] lack the depth required for systems engineering.

Therefore, the necessary integration of architecture and impact knowledge of the different experts requires an approach that supports an easy integration and analysis of domain knowledge that is (i) independent of domain-related vocabularies (neutral), (ii) easy to apply by experts with different levels of qualification and experience (simple), and (iii) facilitates the

Manuscript received July 14, 2026; revised August 12, 2026; accepted July 17, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.148

integration of further knowledge fields if required (extendable).

As a first step to address these concerns, this paper introduces the *System Impact Architecture Model (SIAM) DSL*. The SIAM DSL is a lightweight, structured-text language designed for non-experts in modeling to elicit and validate partial knowledge from diverse groups of domain experts. By integrating a *System Architecture Model* to represent PPR assets and a *System Impact Model* to capture causal paths to desired and undesired outcomes, the language provides a neutral, simple framework to identify risky production paths.

We illustrate the requirements and solution approach using an OAS use case as a specialization of an Industry 5.0 system (cf. Fig. 1). As indicated in [15], both the integrated OAS knowledge and knowledge presentation to the operator have an important impact on an OAS's effectiveness and efficiency. Especially, the tacit and, among experts across disciplines, fragmented knowledge of impact paths within a production system affecting desired and undesired production outcomes shall be integrated into OAS design methods.

Following Design Science [16], we investigate the following research question: ***RQ:*** *To what extent can a simple DSL represent and integrate architecture-impact knowledge emerging from different domain experts in a neutral and extendable way to improve the design and validation of socio-technical assistance systems?*

We build on the *System Impact Architecture Model* [8] for production to define a *SIAM DSL* for representing and validating the socio-technical system architecture with assumed cause-effect impact paths towards desired and undesired production outcomes as input to advanced approaches for engineering, simulation, application, and reuse. We evaluate the feasibility and efficacy of the SIAM DSL through an illustrative case study of gearbox assembly at the *HumTech Lab*,[1] focusing on the multi-domain dependencies between pressing force specifications and production quality.

The remainder of this paper is structured as follows. Section II summarizes related work. Section III introduces the gearbox assembly use case. Section IV introduces the SIAM DSL syntax. Section V reports on a preliminary application case study. Section VI discusses results of the preliminary case study and outlines a future research agenda.

II. Related Work

We review human-centric manufacturing, systems control theory, and software abstraction techniques to establish the baseline for the System Impact Architecture Model DSL.

Operator Assistance System design. Within the Industry 5.0 vision that places the human operator at the center of volatile manufacturing environments, OASs are defined as technical systems that support operators during assembly or manufacturing tasks without replacing them [17]. OASs for assembly systems are part of a generic system architecture [18] (cf. Fig. 1), which consists of (i) the assembly process with its inputs and outputs, (ii) the process environment, (iii) the human operator, (iv) the CPPS to automate the assembly station, and (v) the OAS, a cyber-physical assistance system.

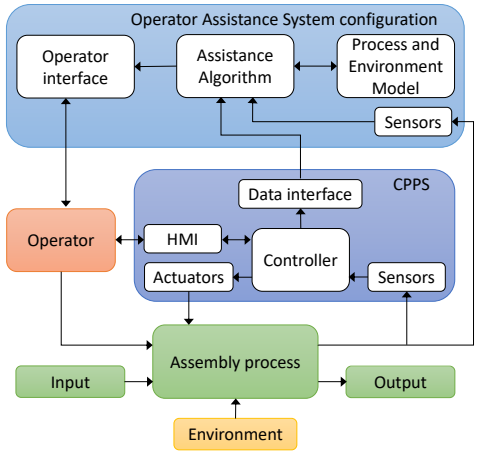


Fig. 1. Generic architecture of assembly and OASs [17], a control system [6].

These systems aim to compensate for human deficits while enhancing existing abilities, primarily through screen-based digital guidelines that reduce operator failures and improve product quality [17], [19]. Therefore, OAS engineers must sufficiently understand the typical control loops of the operator and the CPPS regarding production process performance, to understand what guidance is required or likely to improve operator performance. Traditional OAS design methods [20] exploit product knowledge from product design as well as PPR knowledge [7] from CPPS design.

However, recent work shows that OAS applications increase operator motivation, but frequently fail to reduce cognitive load [21]. A potential cause is insufficient support for quality issue resolution, which stems from a lack of integration of tacit architecture-impact knowledge into the OAS design [20]. An example is the multi-domain relationships between specific pressing forces and the risk of invisible defects. Usually, OAS engineers have only limited access to knowledge on the actual production control loops and need instructions from experts to design sufficiently accurate and complete guidance to the operator, especially based on architecture-impact knowledge related to unintended operator behavior. This work aims to combine PPR knowledge with impact paths to identify open control loops for improving OAS design.

Socio-Technical Hazard Analysis and Control Theory. Industry 4.0 and 5.0 require resilient, self-optimizing systems capable of proactive risk identification. While the Failure Mode and Effects Analysis (FMEA) [22] is the industrial risk assessment baseline, it relies on static, macro-level views. Yet, these are prone to information silos and fail to trace how local technical modifications propagate to business outcomes.

Therefore, this work builds on *STAMP* theory [6], which views socio-technical production as a control system. According to Leveson [6], hazards emerge from dysfunctional control systems, where the mental model views of diverse actors, including operators, CPPS engineers, and OAS designers,

[1] *HumTech Lab* at Otto von Guericke U.: https://tugz.ovgu.de/humtech.html

diverge regarding the assumed and actual runtime states (cf. Fig. 1). These discrepancies create high-impact deviations between expected and actual states, e.g., accidents, low production quality, and low business performance, requiring early-stage elicitation of tacit domain knowledge to identify and close critical open control loops.

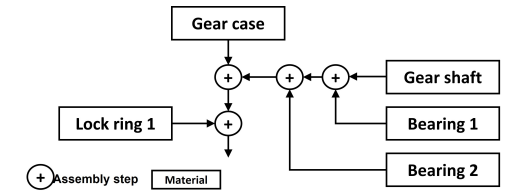
Traditional Architecture and Impact Modeling. Accurate CPPS modeling relies on established frameworks such as RAMI 4.0 [12] and the PPR VDI 3682 [23]. While these frameworks are effective for representing static architectures and hierarchical contexts, they lack the causal logic required for impact prediction, contributing to the architecture-impact gap. Further, current causal models often assume all confounding factors are observed [24], creating a simulation-to-reality gap that obscures physical dependencies in the case of technical changes. To address these limitations, Biffl *et al.* [8] introduced the SIAM meta model to integrate fragmented knowledge in condition networks linked to CPPS architecture. Therefore, this work builds on [8] to design the SIAM DSL for eliciting the knowledge required for facilitating the validation of hypothesized impact paths against production reality.

Foundations for Accessible DSL Design. A DSL offers focused expressive notations tailored to a particular problem domain [25]. Traditionally, a DSL aimed at domain experts to understand, validate, and modify programs in their own terms [26]. Recent DSLs leverage AI to derive engineering and operation artifacts [27]. Unfortunately, heavy DSLs, such as *SysML* [28] or *AutomationML* [7], are often too complex for non-modeling experts to elicit knowledge on production concerns and causal impacts of change, e.g., due to production variants [29] or volatile environments. In contrast, simple structured-text languages such as *Gherkin* (Given-When-Then) are useful to automate monitoring and testing [14], but lack the depth required for advanced systems engineering design. Generative AI has introduced a paradigm shift by enabling dynamic, context-aware low-level code synthesis and modification [27]. However, it has limited effectiveness in making high-level architectural decisions due to reliability concerns.

This paper takes a first step to bridge the architecture-impact gap for OASs by introducing the lightweight structural SIAM DSL. It serves as a boundary object to capture domain experts' contributions on architecture-related causal impact as input to AI-based solution generation, e.g., actionable operator-assistance scenarios to address risks along validated critical process paths with monitoring, documentation, and adaptation.

III. Use Case Operator Guidance Adaptation for a Semi-Automated Assembly Workstation

This section introduces the use case *assemble a gearbox with operator assistance* to illustrate the requirements for bridging the architecture-impact gap during the design and operation of a socio-technical OAS for assembly processes [20]. We abstracted this use case from a domain analysis at the *HumTech Lab*, focusing on the design of an OAS [20] for semi-automated assembly of an abstracted gearbox to evaluate the efficacy of the SIAM DSL for OAS design.



Mounting of gearbox			
1	Pre-mounting of shaft	Clamp shaft	Pick shaft
			Place shaft in pressing unit
			Close clamping
		Press bearing 1 on shaft	Take bearing 1
			Place bearing in press tool
			Press bearing on shaft
		Change shaft clamping	Open clamping
			Pick shaft
			Place shaft in pressing unit
			Close clamping

Fig. 2. Start sequence of a gearbox assembly process as a *gozinto* graph and its related three-level guidance information for a generic gearbox.

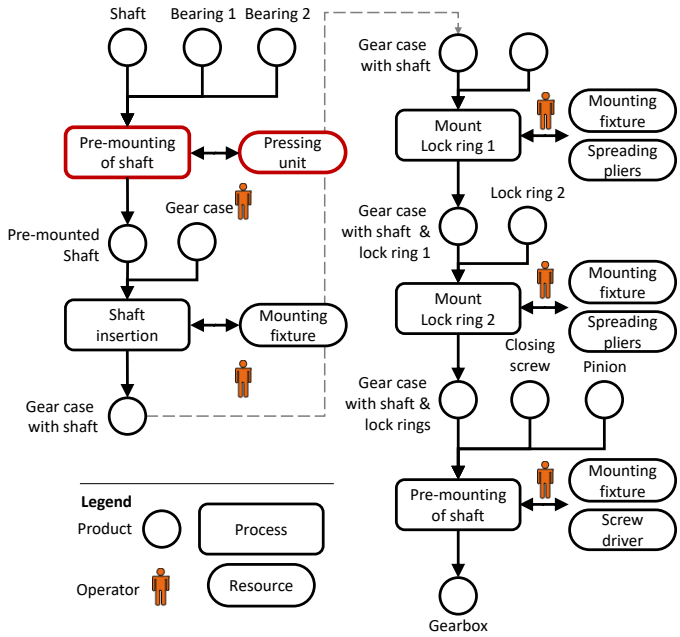


Fig. 3. PPR model [23] for the gearbox assembly process and station.

OAS Design. The workstation is designed for the manual assembly of a generic gearbox comprising a gear case, a shaft with two bearings, two lock rings, and a pinion leading to a combined *Bill of Materials* and a *Bill of Operations* modeled as a *gozinto* graph (cf. Fig. 2, upper part).

The production system design concerns an assembly station, based on a PPR network [23] (cf. Fig. 3), where a *pressing unit* executes a *pre-mounting shaft* process, and the OAS concept regarding operator assistance information by assembly task and associated resources (cf. Fig. 2, bottom). The operator can receive guidance, aligned with operator maturity levels, on three levels of detail: (i) naming the high-level process *pre-mounting of shaft*; (ii) naming process steps like *clamp shaft*; or (iii) detailed handling instructions like *pick shaft*.

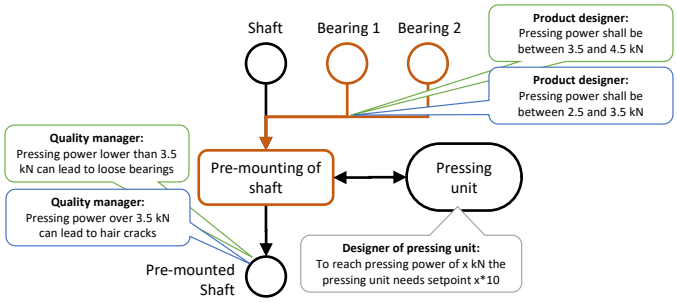


Fig. 4. PPR models [23], [29] on product variability for gearbox assembly.

An OAS, such as the camera-based *Smart Klaus*,[2] supports the process by observing mounting states and providing screen-based digital guidelines. This type of OAS effectively identifies and documents the fulfillment of discrete process steps, such as the successful placement of a shaft. However, it lacks visibility into internal machine parameters, including pressing force parameterization. This results in a divergence between the digital system's state knowledge, the operator's state and view, and the actual technical reality of the assembly.

Tacit partial design knowledge. Unfortunately, critical knowledge required to ensure process quality, such as the pressing force required to press the bearing on the shaft, is often tacit and scattered across stakeholder knowledge silos. For the step *Press a Metal Part*, generalized from the shaft pressing steps, the *Product Designer* holds implicit knowledge on the required force ranges based on material characteristics, e.g., surface structures. Simultaneously, the *CPPS Expert* understands the machine-specific logic required to set the pressing unit's parameters to achieve these setpoints. The *Quality Manager* (QM) identifies the causal outcomes of force deviations: a pressing force below thresholds, e.g., 3.5 kN, results in loose bearings, a failure easily detectable in quality control. However, a force exceeding the threshold leads to invisible hair cracks that escape standard inspection and cause product failure at the customer (cf. Fig. 4). The QM knows that different product variants call for different parameter settings. For example, gearbox variants A and B may use bearings of different materials, requiring distinct force thresholds, e.g., 2.5–3.5 kN vs. 3.5–4.5 kN, which pose hazards if operators do not adjust the force during product changes. If operators fail to adjust the pressing force for a specific variant, the resulting *Low-quality pressed metal part* represents a high-impact deviation between intended design and runtime state. Such implicit knowledge is not available to the *operator* of the assembly station, unless the knowledge is collected and integrated in the OAS. The SIAM DSL shall facilitate the elicitation and integration of the relevant knowledge about potential risks and represent the involved assets and links to their characteristics (cf. Fig. 4) to facilitate their evaluation in relation to intended and unintended operator behavior.

Requirements. For the SIAM DSL, we derived four essential *modeling requirements* from the use case: (R1) *System architecture model representation*, based on a production model (i) utilizing PPR architecture elements and (ii) identifications of variants and versions; (R2) *System impact model representation*, to connect conditions in the architecture with process outcomes; (R3) *Stakeholder knowledge area representation*, to document required stakeholders with access to knowledge and data; and (R4) *Modeling pragmatics* of the DSL to ensure accessibility to domain experts that are non-experts in modeling, (i) simplicity and domain-neutrality, (ii) extensibility for adaptation to specific domains, and (iii) usability in typical modeling contexts, e.g., easy to read, write, and process in a workshop, considering AI support, to provide input to advanced control system modeling and analysis.

[2]*Smart Klaus* OAS: https://www.optimum-gmbh.de/en/

IV. System Impact Architecture Model DSL

To address the requirements collected in Section III, this section introduces the syntax of the SIAM DSL elements with examples (cf. Tabs. I to IV), based on the SIAM meta model [8]. To address requirement R1, representation of the *system architecture model* (cf. Fig. 5), Tabs. I to II define assets, asset network relations, and asset characteristics. To address requirement R2, representation of the *system impact model* (cf. Fig. 6), Tab. III defines conditions and condition impact paths. To address requirement R3, representation of *stakeholders with access to knowledge and data*, Tab. IV defines stakeholder knowledge areas, applicable both for assets and conditions. To address requirement R4, *modeling pragmatics*, the definition of the DSL is kept minimal to elements that participants actively used in modeling workshops to elicit domain knowledge, allowing for extending the DSL as required for specific applications, e.g., adapting/adding types, markers, or new syntax elements.

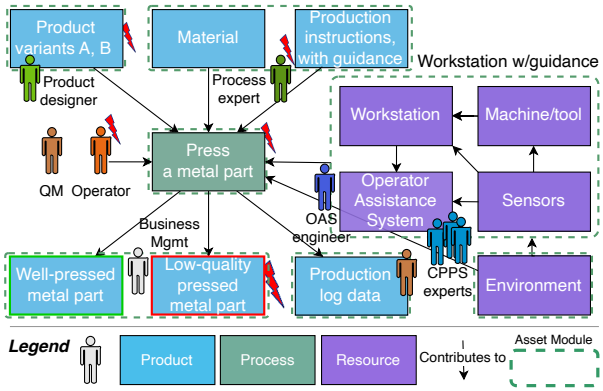


Fig. 5. System architecture model: process with CPPS and OAS (cf. Lst. 1)

The *Asset* (cf. Tab. I), in production a PPR asset [7], is the core of the system architecture model (cf. Fig. 5), identified by a name, unique ID with consideration of asset variants and versions, and an extensible type system. Markers provide a means to annotate assets with metadata for graph queries on the system architecture model, e.g., critical elements or assets that require validation.

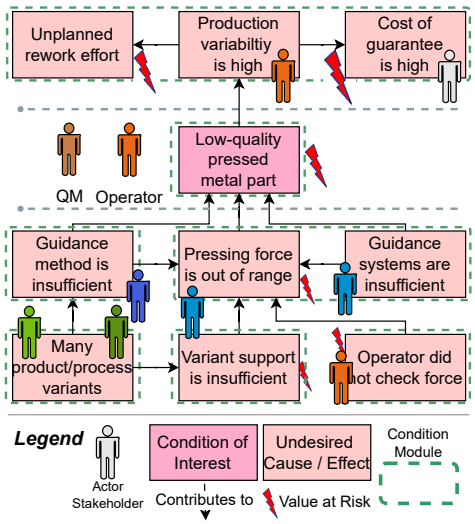


Fig. 6. System impact paths: undesired conditions (cf. Lst. 2).

TABLE I
SIAM DSL: ARCHITECTURE: ASSET NETWORK SYNTAX (CF. FIG. 5).

Syntax Rule	ASSET <name>, <id>, <type>, [<markers>].
<name>	Quoted string (human-readable label)
<id>	Unique ID with embedded variant/version
<type>	Dot notation set (e.g., product, process, resource.system)
[<markers>]	Optional list of symbolic tags in brackets
ASSET "Press a Metal Part", PROC-001v01, process, [critical].	
ASSET "Material", PROD-001v01, product, [to_validate].	
ASSET "Low-quality pressed metal part", PROD-002v01, product, [defect].	
Syntax Rule	CONTRIBUTES_TO <asset1>, <asset2>, <asset3>, ...
<asset>	Name or ID of an existing asset
Impact Path	Assets contribute sequentially: asset1 → asset2 → ...
Inputs Tuple	Grouped source assets: (asset1, asset2), asset3
Outputs Tuple	Grouped target assets: asset1, (asset2, asset3)
# — Simple Path and Direct Chains —	
CONTRIBUTES_TO "Operator", "Press a Metal Part". # cf. Fig. 5	
# — Combined Input Assets —	
CONTRIBUTES_TO ("Orders for Product Variants A, B", "Production Instructions with Guidance", "Material"), "Press a Metal Part".	
# — Combined Output Assets —	
CONTRIBUTES_TO "Press a Metal Part", ("Well-pressed Metal Part", "Low-quality Pressed Metal Part", "Production Log Data").	

The asset network relation *Contributes_to* (cf. Tab. I), connects assets to an asset network (cf. Fig. 5), a directed graph, by indicating that an asset contributes to a related asset, e.g., an operator, product, or system to a process. At its simplest, this relation connects two assets, identified by name or ID. For writing convenience, the SIAM DSL provides options to describe an asset impact path, or to group several input assets or output assets.

The *asset characteristic* (cf. Tab. II), refines an asset and provides a connection point for conditions that refer to an asset characteristic. The characteristic refers to its asset by name and ID, is identified by a locally unique name and version, and refers to a unit. The threshold vector holds a list of ascending key-value pairs that conditions can refer to,

TABLE II
SIAM DSL: ASSET CHARACTERISTICS SYNTAX AND EXAMPLES.

Syntax Rule	ASSET_Characteristic <Asset_Name>, <Asset_ID>; <Characteristic_Name>; <Characteristic_ID>; unit <Unit> [; thresholds (<label>: <value>; ...)].
<Asset_...>	Name and ID of an existing asset
<Characteristic_...>	Descriptive name and version identifier
<Unit>	Standard unit or custom unit (e.g., kN, s)
thresholds	Ascending label/value vector configuration
ASSET_Characteristic "Press a Metal Part", PROC-001v01; "Cycle time"; V02; unit s; thresholds (min: 5.0; normal: 10.0; max: 30.0).	

e.g., *pressing force is high* refers to the threshold *high* of the characteristic *pressing force* that belongs to an asset, such as a product, process, or system.

TABLE III
SIAM DSL: CONDITION NETWORK SYNTAX (CF. FIG. 6).

Syntax Rule	CONDITION "<statement>", <id>, <type>, <asset_ref>, [<markers>].
<statement>	Free-form quoted natural language condition
<id>	Unique ID with variant/version (e.g., UCON-011v01)
<type>	Hierarchical type system, e.g., *condition.undesired*
<asset_ref>	Name or ID of an existing asset or characteristic
[<markers>]	Optional list of symbolic tags in brackets
CONDITION "Pressing force is out of range", UCON-008v01, condition.undesired, "Press a Metal Part", [critical, convergence_point].	
CONDITION "Cycle time is within limits", DCON-212v01, outcome.desired, "Press a Metal Part", [to_validate].	
Syntax Rule	CONDITION_CONTRIBUTES_TO <condition1>, <condition2>, <condition3>, ...
<condition>	Statement/name or ID of an existing condition
Impact Path	Conditions contribute sequentially: c1 → c2 → ...
Inputs Tuple	Grouped source conditions: (c1, c2), c3
Outputs Tuple	Grouped target conditions: c1, (c2, c3)
# — Simple Path and Direct Chains —	
CONDITION_CONTRIBUTES_TO "Low-quality pressed metal part", "Production variability is high", "Cost of guarantee is high".	
# — Combined Input Conditions —	
CONDITION_CONTRIBUTES_TO ("Pressing force is out of range", "Guidance method is insufficient", "Guidance systems are insufficient"), "Low-quality pressed metal part".	
# — Combined Output Conditions —	
CONDITION_CONTRIBUTES_TO "Product/process variance is high", ("Variant support is insufficient", "Guidance method is insufficient").	

The *Condition* (cf. Tab. III) is the core of the system impact model (cf. Fig. 6), identified by a statement, unique ID with consideration of condition versions, and an extensible type system. A condition shall refer to an existing asset or characteristic to ensure a valid connection between the system architecture and impact models as input to bridge the architecture-impact gap. Markers provide the means to annotate a condition with meta information for graph queries on the system impact model, e.g., critical conditions, or conditions that require validation or improvement.

The condition impact path relation *Condition_Contributes_-to* (cf. Tab. III), connects conditions to a condition path or network (cf. Fig. 6), a directed graph, by indicating that a

condition contributes to a condition, e.g., pre-condition to a post-condition. In the simplest form, this relation connects a list of conditions, referred to by statement/name or ID. For writing convenience, the SIAM DSL provides options to describe a condition impact path, or to group several input conditions or output conditions.

TABLE IV
SIAM DSL: STAKEHOLDER KNOWLEDGE AREA SYNTAX (CF. FIGS. 5/6).

Syntax Rule	PART_OF <stakeholder area> CONTAINS <stk1>, ... ; <item1>, <item2>, ...
<stk. area>	Quoted string (human-readable label)
<stk>, ...	List of stakeholders (asset names)
<item>, ...	Quoted name/ID of an existing asset or condition

 # — Asset Areas —
PART_OF "Output Concerns" CONTAINS "Quality Manager";
"Well-pressed Metal Part", "Low-quality Pressed Metal Part".
 # — Condition Areas —
PART_OF "Process Concerns" CONTAINS "Operator", "Quality Manager";
"Product/process variance is high", "Pressing force is out of range".

The stakeholder area relation *Part_of ... Contains* (cf. Tab. IV), describes for a stakeholder area the stakeholders and assets or conditions, for which these stakeholders have access to knowledge or data (cf. Figs. 5 and 6). The relation shall refer to existing assets (including stakeholders) and conditions.

V. CASE STUDY ON THE APPLICATION OF THE SIAM DSL

We conducted a preliminary case study for the use case *gearbox assembly* to evaluate the application of the SIAM DSL with the requirements for a SIAM DSL. It was selected as a typical case, according to Runeson *et al.* [30], as the multi-domain interaction between pressing force specification, application, and validation is representative of high-value dependencies common in semi-automated assembly with assistance. Two researchers facilitated and observed a three-step SIAM workshop involving three domain experts who represented the roles Quality Manager (QM), OAS engineer, operator, and CPPS experts (cf. Section III). In the one-day workshop, participants and facilitators used the SIAM DSL to document architecture and impact models and gain insight on the pragmatic quality of the SIAM DSL.

The preliminary SIAM DSL application consisted of three steps: (1) Scoping of production with hazards and preliminary design of the system architecture model; (2) Socio-technical system architecture and impact analysis with domain experts; and (3) Specification of scenarios for operation assistance to prevent or mitigate production hazards due to the wrong pressing force. The case study yielded the following results.

(1) Scoping and design. The quality manager, as a proxy for the solution architect, defined the scope of work as the use case *assemble a gearbox with operator assistance*, with a focus on the process *Pre-mount shaft* with the critical step *Press a Metal Part*, which is representative of a wide range of similar assembly processes. From the PPR model (cf. Fig. 3), the QM derived the technical part of the architecture, shown in Fig. 5, illustrating the architecture of the socio-technical production system around the process step *Press a metal part*: input products, process, output products, and production systems required for automating production and operator assistance. The figure illustrates the key stakeholders with diverse knowledge on the production assets required to understand conditions that contribute to production outcomes.

(2) System architecture and impact analysis. To warm up, the QM and the domain experts followed the process *Pre-mount shaft* to understand the context of and contributions to the process step *Press a Metal Part* towards the desired outcome *Well-pressed metal part*. In parallel to the warm-up session, a facilitator provided LLMs (*Gemini* and *Qwen 3.5*) with the definition of the SIAM DSL and the PPR model (cf. Fig. 3, assets and asset network, without stakeholders) to extract the system architecture model in SIAM DSL (cf. Lst. 1). The domain experts validated the translation of the graphic model to the SIAM DSL, which was mostly correct.

 # — Process Assets ———————————
ASSET "Press a Metal Part", PROC-001v01, process, [bolt].
 # — Output & Input Assets ———————————
ASSET "Well-pressed Metal Part", OUT-001v01, product, [critical].
ASSET "Low-quality Pressed Metal Part", OUT-002v01, product, [defect].
ASSET "Production Log Data", DATA-002v01, data.
ASSET "Production Instructions with Guidance", DATA-001v01, data, [bolt].
ASSET "Orders for Product Variants A, B", DATA-003v01, data.
 # — Resource Assets ———————————
ASSET "Operator", HUM-001v11, resource.human.
ASSET "Guidance System", SYS-001v01, resource.system.
ASSET "Sensors", SYS-002v01, resource.system.
ASSET "Workstation", SYS-003v01, resource.system.
ASSET "Machine/Tool", SYS-004v01, resource.system.
 # — Asset Network ———————————
CONTRIBUTES_TO "Operator", "Press a Metal Part".
CONTRIBUTES_TO ("Orders for Product Variants A, B",
"Production Instructions with Guidance", "Material"), "Press a Metal Part".
CONTRIBUTES_TO ("Workstation", "Machine/Tool", "Guidance System"),
"Press a Metal Part".
CONTRIBUTES_TO "Press Metal Part", ("Well-pressed Metal Part",
"Low-quality Pressed Metal Part", "Production Log Data").
 # — Asset Characteristics: Pressing Force (Product A vs B) —
ASSET_Characteristic "Press a Metal Part", PROC-001v01;
"Pressing force"; V12; unit kN;
thresholds (min: 0.0; low: 2.5; high: 3.5; max: 5.0).
ASSET_Characteristic "Press a Metal Part", PROC-001v01;
"Pressing force"; V13; unit kN;
thresholds (min: 0.0; low: 3.5; high: 4.5; max: 5.0). # cf. Section III

Listing 1: SIAM DSL: Asset Network Architecture (cf. Fig. 5).

Then, the QM and the domain experts explored direct and indirect causes that contribute to the undesired production outcome *Low-quality pressed metal part*. First, they inverted desired contributions along the sun-shine path to identify undesired conditions. Then they reasoned backward from the outcome *Low-quality pressed metal part* along the edges of the asset network (cf. Fig. 5) to systematically identify undesired conditions, resulting in a candidate list.

In parallel, a facilitator asked the LLMs for main causes of low-quality pressing and received a list of conditions with explanations. The domain experts discussed these candidate cause conditions and integrated new conditions into their list of conditions to validate.

Then they validated the conditions with the assets and identified the asset characteristics in Lst. 1 to represent different versions of thresholds for pressing force for product variants A and B with different material characteristics (cf. Fig. 4).

```
# — Conditions ————————————————
CONDITION "Pressing force is out of range", UCON-008v01,
condition.undesired, "Press a Metal Part", [critical, convergence_point].
CONDITION "Operator did not check force", UCON-004v01,
condition.undesired, "Operator", [critical].
CONDITION "Production variability is high", UCON-006v01,
condition.undesired, "Press Metal Part", [critical].
CONDITION "Pressing force is high", UCON-012v01,
condition.undesired, "Press a Metal Part".
# — Condition Impact paths ————————————
CONDITION_CONTRIBUTES_TO "Low-quality pressed metal part",
"Production variability is high", "Cost of guarantee is high".
CONDITION_CONTRIBUTES_TO ("Pressing force is out of range",
"Guidance method is insufficient", "Guidance systems are insufficient"),
"Low-quality pressed metal part".
CONDITION_CONTRIBUTES_TO ("Operator did not check force",
"Variant support is insufficient", "Guidance method is insufficient",
"Guidance systems are insufficient"), "Pressing force is out of range".
CONDITION_CONTRIBUTES_TO "Many product/process variants",
("Variant support is insufficient", "Guidance method is insufficient").
```

Listing 2: SIAM DSL: Condition Impact Paths (cf. Fig. 6).

Further, they identified impact paths (cf. the condition impact paths in Lst. 2), resulting in a list of main undesired impact paths. Shared conditions across several impact paths resulted in a network of main undesired impact paths (cf. Fig. 6) that the domain experts sketched on a whiteboard to understand visually the interaction of main impact paths.

Fig. 6 shows, for the undesired condition of interest, *Low-quality pressed metal part*, impact paths in the socio-technical production system that contribute to this condition. These impact paths follow the architecture impact paths (cf. Fig. 5).

Further, these impact paths concern knowledge of diverse stakeholders with access to engineering and runtime data to validate hypotheses about impact paths. Therefore, the domain experts added the stakeholder areas to both the asset and condition networks to validate a stakeholder role for each model element. They tracked the stakeholder areas in a table, consistent with the SIAM DSL (cf. Lst. 3). Lst. 3 represents the stakeholder areas for the asset network (cf. Fig. 5) as input to propagate these stakeholders to related conditions in the condition network (Fig. 6). In the workshop, this activity revealed model elements without stakeholders, a major source of risk to architecture-impact quality, and consequently, the introduction of new stakeholders, e.g., business management.

(3) OAS scenario specification. The SIAM DSL conditions provided input for the QM and OAS engineer to specify OAS behavior on critical paths, e.g., production states to observe and operator assistance to provide. Lst. 4 illustrates a feature in *Gherkin* [14] for operating assistance for *pressing a metal part with product type variation*, to warn the operator (cf. lines in violet color) after a change of product type, a production state that is prone to error, and to collect operator feedback.

Therefore, the workshop provided preliminary evidence for the pragmatic use of the SIAM DSL as a lightweight *hub DSL* to collect, integrate, and exchange architecture and impact

```
# — Asset Areas ———————————————
PART_OF "Process Concerns" CONTAINS "Operator", "Quality Manager";
"Press a Metal Part". # cf. Fig. 5 and Tab. IV
PART_OF "Output Concerns" CONTAINS "Quality Manager";
"Well-pressed Metal Part", "Low-quality Pressed Metal Part".
PART_OF "Input Concerns" CONTAINS "Quality Manager"; "Material",
"Orders for Product Variants A, B", "Production Instructions".
PART_OF "Systems" CONTAINS "CPPS expert";
"Workstation", "Guidance System", "Sensors", "Machine/Tool".
# — Condition Areas ———————————————
PART_OF "Process Concerns" CONTAINS "Operator", "Quality Manager";
"Product/process variance is high", "Pressing force is out of range".
```

Listing 3: SIAM DSL: Selected stakeholder knowledge areas.

```
Feature: Press a metal part with product type variation
As an assembly workstation operator,
I want to produce a high share of good metal parts, for varying product types,
So that the share of NOK gearboxes is low And the OEE is high.
Background:
Given an order for batches of several gearbox types # in production history
Given sufficient input materials of good quality
Given the assembly workstation and the operator assistance system work well
Given the operator is trained well # but may have more or less experience
@SadPath @High Waste in Production
Scenario: damage due to wrong force setting after product type change
Given the product type changed recently # compare input product with history
When the operator does not check the correct parameter value
When the operator presses the metal part with the assembly workstation
Then the share of bad metal parts is high # 95%, in-process or output
@Operator Assistance @AdaptationPath @Risk-Waste Reduction
Scenario: assemble after product type change
Given the product type changed recently # compare input product with history
When the operator assistance warns with the correct parameter value
When the operator sets, checks, and confirms the correct parameter value
When the operator presses the metal part with the assembly workstation
Then the share of good metal parts is high # 95%, in-process or output
Then the share of gearboxes NOK is low # 5%, inspection of output
```

Listing 4: Feature: Adaptive operator assistance instructions.

knowledge collected in different formats for validation, integration, and as input to efficiently design new model versions based on previous model parts in the SIAM DSL.

VI. Discussion and Conclusion

This section evaluates the preliminary research results concerning the research question (RQ) *to what extent a simple DSL can represent and integrate architecture-impact knowledge emerging from different domain experts in a neutral and extendable way.* The application of the SIAM DSL demonstrates that a lightweight, structured-text language can effectively contribute to bridging the architecture-impact gap: by documenting validated architectural paths that support assumed impact logic, the DSL provides a simple framework to identify risky production states that traditional, static modeling frameworks often miss. In the gearbox assembly use case, the SIAM DSL facilitated the elicitation of tacit and scattered domain knowledge that had previously been isolated in the silos of product design, CPPS engineering, and QM.

Preliminary assessment. The SIAM DSL (i) by design represents asset networks (R1), condition impact paths (R2), and stakeholder knowledge areas (R3); (ii) is simple as it focuses on the required elements, described in a structured language, in the core close to the natural language that domain

experts use for naming assets, conditions, and impact paths; (iii) is pragmatic and minimal as the elements asset network, impact paths, and stakeholder areas are required to clarify valid solutions or open issues; (iv) is flexible and extensible to take up new asset types, characteristics, variants, and versions as required in the discussion as input to assessment and comparison, e.g., of local architecture views.

Lessons Learned. Several insights were derived from the feasibility study at the *HumTech Lab*. *Pragmatic integration*: The SIAM DSL served as a robust boundary object that unified inputs from spoken language, tables, and whiteboard sketches into a structured, textual representation suitable for both human experts and machine agents, enabling quick capture of elusive expert contributions. *AI-Readiness and Efficiency*: LLM functions proved instrumental in effectively and efficiently transforming human shorthand into complete SIAM DSL syntax while validating consistency across multi-domain inputs. *Exposing Open Control Loops*: Following STAMP theory, the DSL revealed critical open control loops where the traditional OAS lacked visibility into internal machine parameters, such as pressing force for product variants.

Limitations. The SIAM DSL is designed primarily for the elicitation and representation of knowledge in workshop settings. It is not intended to replace heavy systems engineering or simulation languages. In addition, the empirical results are subject to potential researcher and selection bias inherent in case studies derived from a standard production process.

Conclusion. The SIAM DSL provides a neutral, simple, and extendable framework for bridging the Architecture-Impact Gap in socio-technical CPPSs. By linking asset characteristics to condition impact paths, the language enables non-modeling experts to specify and validate multi-domain dependencies required for making OASs resilient to volatile environments.

Research agenda. *Empirical evaluation of the SIAM DSL*: We plan to strengthen the evaluation with PPR variability models to derive OAS configuration artifacts for mitigating risks of high-mix production variants. *Explore architecture-impact gap variants*: To identify and close open control loops, we plan to investigate to what extent scenarios can reveal architecture-impact gaps by documenting design-time intents and monitoring run-time states [31] as input to quality data and gap analysis. *Scaling of workshop results*: We plan to explore AI functions for the efficient integration of SIAM DSL artifacts into large CPPS and library models.

References

[1] M. Abramovici and O. Herzog, *Engineering im Umfeld von Industrie 4.0*. Acatech, Utz Verlag, 2016.

[2] B. Vogel-Heuser, T. Bauernhansl, and M. Ten Hompel, *Handbuch Industrie 4.0 Bd. 4*, 2nd ed. Springer, 2017.

[3] Y. Lu, H. Zheng *et al.*, "Outlook on human-centric manufacturing towards industry 5.0," *Jour. Manuf. Sys.*, vol. 62, pp. 612–627, 2022.

[4] C. Bechinie, S. Zafari, L. Kroeninger, J. Puthenkalam, and M. Tscheligi, "Toward human-centered intelligent assistance system in manufacturing: challenges and potentials for operator 5.0," *Procedia Computer Science*, vol. 232, pp. 1584–1596, 2024, int. Conf. Industry 4.0 and Smart Manuf.

[5] G. Garcia-Garcia, Y. Singh, and S. Jagtap, "Optimising changeover through lean-manufacturing principles: A case study in a food factory," *Sustainability*, vol. 14, no. 14, 2022.

[6] N. G. Leveson, *Engineering a safer world: Systems thinking applied to safety*. The MIT Press, 2016.

[7] M. Schleipen and R. Drath, "Three-view-concept for modeling process or manufacturing plants with AutomationML," in *IEEE ETFA*, 2009.

[8] S. Biffl, H. Rahmani, K. Meixner, D. Hoffmann, and A. Lüder, "Value-based Change Impact Analysis in Production with System Impact Architecture Modeling," in *Proc. IEEE Int. Conf. ETFA (in press)*, 2026.

[9] C. Di Ciccio, A. Marrella, and A. Russo, "Knowledge-intensive processes: Characteristics, requirements and analysis of contemporary approaches," *Jour. Data Semantics*, vol. 4, no. 1, pp. 29–57, Apr. 2014.

[10] S. Biffl, A. Lüder, and D. Gerhard, *Multi-Disciplinary Engineering for Cyber-Physical Production Systems: Data Models and Software Solutions for Handling Complex Engineering Projects*. Springer, 2017.

[11] A. Lüder, D. Hoffmann, P. Hünecke *et al.*, "Entwicklung einer informationslogistik zur automatisierung des auomatisierungsengineering," in *Entwurf Komplexer Automatisierungssysteme (EKA)*, vol. 1, 2026.

[12] A. Shirbazo, B. Li, S. Ata, H. L. Ramandi, and S. Saydam, "A guideline for the standardisation of smart manufacturing and the role of rami 4.0 in digitising the industrial sector," *IEEE Internet of Things Journal*, 2025.

[13] D. Moody, "The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering," *IEEE Transactions on software engineering*, vol. 35, no. 6, pp. 756–779, 2009.

[14] M. S. Farooq, U. Omer, A. Ramzan, M. A. Rasheed, and Z. Atal, "Behavior driven development: A systematic literature review," *IEEE Access*, vol. 11, pp. 88 008–88 024, 2023.

[15] C. Stockinger, L. Polanski-Schräder, and I. Subtil, "The effect of information level of digital worker guidance systems on assembly performance," *Applied Ergonomics*, vol. 106, p. 103896, 2023.

[16] R. J. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014.

[17] M. König and H. Winkler, "Investigation of assistance systems in assembly in the context of digitalization: A systematic literature review," *Journal of Manufacturing Systems*, vol. 78, pp. 187–199, 2025.

[18] H. Oestreich, M. Heinz-Jakobs, P. Sehr, and S. Wrede, *Human-Centered Adaptive Assistance Systems for the Shop Floor*. Cham: Springer International Publishing, 2023, pp. 83–125.

[19] M. König and H. Winkler, "Empirische ergebnisse zum einsatz von assistenzsystemen in der montage," *Zeitschrift für wirtschaftlichen Fabrikbetrieb*, vol. 119, no. 7-8, pp. 563–568, 2024.

[20] B. Pokorni, D. Popescu, and C. Constantinescu, "Design of cognitive assistance systems in manual assembly based on quality function deployment," *Applied Sciences*, vol. 12, no. 8, 2022.

[21] M. Walczok and T. Bipp, "Investigating the effect of intelligent assistance systems on motivational work characteristics in assembly," *Journal of Intelligent Manufacturing*, vol. 35, pp. 1949–1962, 2022.

[22] DIN60812, *DIN EN 60812:2015-08: Failure Mode and Effects analysis (FMEA)*, International Standard, DIN EN Std. 60 812:2015–08, 2015.

[23] *VDI Guideline 3682 Formalised Process Descriptions*, Beuth Std., 2015.

[24] E. A. Akinluyi, K. Ison, and P. J. Clarkson, "Mapping outcomes in quality improvement and system design activities: the outcome identification loop and system impact model," *BMJ Open Quality*, vol. 8, no. 3, 2019.

[25] A. Van Deursen *et al.*, "Domain-specific languages: An annotated bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[26] S. Greiner, B. Wiesmayr, K. Feichtinger, K. Meixner *et al.*, "Maturity evaluation of domain-specific language ecosystems for cyber-physical production systems," in *Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2023, pp. 1–8.

[27] M. Schieseck, P. Topalis, and A. Fay, "A graphical modeling language for artificial intelligence applications in automation systems," in *Int. Conf. on Ind. Inf. (INDIN)*. IEEE, 2023, pp. 1–7.

[28] L. Beers, H. Nabizada, M. Weigand, F. Gehlhoff, and A. Fay, "A sysml profile for the standardized description of processes during system development," in *Int. Sys. Conf. (SysCon)*. IEEE, 2024, pp. 1–8.

[29] K. Meixner, K. Feichtinger, H. S. Fadhlillah, S. Greiner, H. Marcher, R. Rabiser, and S. Biffl, "Variability modeling of products, processes, and resources in cyber–physical production systems engineering," *Journal of Systems and Software*, vol. 211, p. 112007, May 2024.

[30] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines*. John Wiley & Sons, 2012.

[31] S. Biffl, M. Martinelli, H. Rahmani, and M. Picone, "Business intelligence architecture for process quality monitoring with bdd," in *Proc. Software Quality Days (SWQD 2026)*. Springer LNCS, 2026, pp. 1–22.

Large Language Models for Software Architecture Design Support in Self-Adaptive Systems: Early Insights from an Exploratory Systematic Review

Nadeem Abbas
Linnaeus University, Växjö, Sweden
nadeem.abbas@lnu.se

Nazia Shahzadi
Linnaeus University, Växjö, Sweden
nazia.shahzadi@lnu.se

Abstract—**Modern computing systems exhibit increasing heterogeneity and often require runtime self-management and adaptation to cope with their structural and operational complexity, as well as changes in their environment and requirements. *Self-Adaptive Software Systems (SASS)* represent a class of context-aware and autonomous systems designed to manage such complexity. However, designing such systems remains challenging due to their complexity, runtime variability, and the continuous need to ensure functional and quality requirements. *Large Language Models (LLMs)* and *Generative AI (Gen AI)* offer promising capabilities, yet their use in the architectural design of SASS remains poorly understood. To that end, this study reports a *work in progress* systematic review. The review findings reveal that the use of LLMs and other Gen AI approaches for the architectural design of SASS remains nascent, with only *four* relevant studies identified. Across these studies, LLMs act as augmentative reasoning components, concentrated in the monitoring, analysis, planning, and knowledge phases of the MAPE-K loop and are only partially present in execution. Characteristics such as hybrid architectures, multi-agent reasoning, and retrieval-augmented grounding recur across the reviewed studies; however, given the small and heterogeneous evidence base, these are best viewed as preliminary observations rather than established trends, and trustworthiness and runtime assurance remain underexplored. As a work in progress, this paper contributes an initial characterization of LLM-supported design in self-adaptive systems, outlines research directions, and aims to stimulate discussion within the community on advancing LLM-supported architectural design for self-adaptive and autonomous software systems.**

Index Terms*—*Large Language Models, Self-Adaptive Software Systems, Software Architecture, Architectural Design, Architectural Reasoning, MAPE-K, Autonomous Computing Systems, Systematic Literature Review

I. INTRODUCTION

Modern computing systems increasingly operate at scale across the cloud–edge continuum, within rapidly evolving ecosystems with heterogeneous components, technologies, environments, and requirements. Their structural and operational complexity often exceeds what manual configuration or static control logic can manage, particularly when timely decisions must be made under strict quality-of-service constraints. These conditions place growing cognitive demands on software architects, who must balance quality attributes, evaluate alternatives, and reason over large design spaces under uncertainty [1], [2]. Architectural design[1] is inherently complex: architects must navigate combinatorial design spaces, evaluate multiple design options, and manage multi-objective trade-offs under given constraints. The architectural design becomes even more complicated in self-adaptive software systems and their product lines [3], [4].

Self-Adaptive Software Systems (SASS) are software systems capable of autonomously adapting their behavior and configuration at runtime in response to changes in requirements, operating environments, and system conditions [5]. The rapid emergence of agentic AI and increasing connectivity of Cyber-Physical Systems (CPS) introduces highly dynamic operating environments in which system goals, requirements, and execution contexts may evolve unpredictably during runtime. CPS tightly couple software with continuously changing physical environments through sensing, actuation, human interaction, and communication with other systems, resulting in persistent uncertainty and environmental variability. All such characteristics of modern software systems require self-adaptation as a fundamental architectural capability rather than an optional feature [6].

Consequently, software systems must be capable of continuously adapting both their functional behavior and quality attributes to maintain correctness, reliability, and performance under changing conditions. These evolving demands significantly increase the complexity of architectural design. Designing SASS[2] and their product lines requires architects to reason about multiple quality attributes and adaptation objectives while managing several dimensions of variability, including domain, runtime, and cross-domain variability [3], [4]. Collectively, these challenges substantially expand the architectural design space, making the architectural design of SASS an inherently complex task that demands sophisticated

[1]From this point onward, we use the term architectural design as an umbrella concept encompassing architectural analysis, reasoning, decision-making, and related activities performed during software architecture design.

[2] From this point onward, we use SASS to denote both individual self-adaptive systems and their product lines unless stated otherwise.

Manuscript received July 14, 2026; revised July 18, 2026; accepted July 17, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.149

architectural reasoning and decision-making. Large Language Models (LLMs)[3] [7], as a form of generative and increasingly agentic AI, have emerged as a promising technology for augmenting architectural reasoning and decision-making, with growing evidence of their value across software engineering (SE) tasks [8], [9]. Their ability to interpret heterogeneous context, synthesize design alternatives, and explain rationale aligns closely with the reasoning-intensive activities that SASS demand [10], [11]. Yet how LLMs can systematically support architectural design, analysis, and reasoning specifically for SASS, as opposed to general software systems, remains poorly understood, motivating the review reported in this paper.

Gap: Despite growing use of LLMs and Gen AI for SE, little is known about how these can be used and effectively integrated into the architectural design of SASS, particularly for trade-off analysis and reasoning support required by architects of complex, large scale, and highly heterogeneous SASS.

Goal and Research Questions (RQs): This study aims to address the above gap by reporting an ongoing Systematic Literature Review (SLR) and early insights gained from the review. The review is guided by the following RQs:

- **RQ1:** *What has been reported to date for the use of LLMs and Gen AI for architectural design support in SASS?*

  Rationale: This question maps the state of the art by identifying existing studies on LLM and Gen AI-supported architectural design in SASS.

- **RQ2:** *What techniques are reported to optimize and evaluate LLMs and Gen AI for architectural design support in SASS?*

  Rationale: The support offered by LLMs and Gen AI is valuable only if results are accurate, grounded, and trustworthy; analyzing optimization and evaluation techniques reveals whether the reported LLM-based architectural design support methods meet these needs.

- **RQ3:** *What are the emerging trends in the use of LLMs and Gen AI for architectural design in the development of SASS?*

  Rationale: Understanding emerging trends helps anticipate innovations and guide SASS design tools and frameworks.

Guided by these RQs, the study yields the following contributions: **(1)** characterization of the role of LLMs in architecture design in early stage, and to our knowledge, the first SLR at the intersection of LLMs, architectural design, and SASS; **(2)** a consolidated view of the role of LLMs in the software architecture design process along the MAPE-K loop, spanning design-time and runtime activities; **(3)** identification and analysis of techniques reported to optimize and evaluate LLMs for architectural design support in the design of SASS; **(4)** a preliminary research agenda on how to use LLMs for architectural design support in SASS; and **(5)** the identification of 58 studies that target use of LLM for software architecture design for future analysis of their integration for architectural design support for SASS.

The early results show that the use of LLMs for architecture design support in SASS is very limited and largely underexplored. Recurring characteristics such as Retrieval-Augmented Generation (RAG), multi-agent orchestration, and feedback-driven refinement are observable across the selected studies, although the small and heterogeneous sample does not yet permit conclusions about broader trends in the field. Key challenges remain on the research agenda, including standardized evaluation, assurance, variability support, and architecture-centric integration across the MAPE-K loop.

The rest of the paper is organized as follows: Section II describes the research method; Section III reports results; Section IV synthesizes answers to the RQs and discusses implications; Section V examines threats to validity; Section VI describes what we plan for the future work; and Section VII concludes the study.

II. METHOD

The study goal and RQs, specified in Section I, require a review of published literature. We therefore used the SLR method and followed the guidelines of Kitchenham et al. [12], to systematically identify and analyze relevant studies. The review process is detailed in the following subsections.

A. Review Planning and Execution

To ensure methodological rigor and consistency, and reduce researcher bias, we developed a review protocol with well-defined RQs, search strategy, inclusion and exclusion criteria, quality assessment, and data extraction and analysis methods. We applied the protocol and used automatic search, manual screening, and snowballing [13] to conduct the review. We built a structured search string (Listing 1) using keywords derived from the RQs. We validated the search string using the Quasi–Gold Standard (QGS) [14] method to ensure effective retrieval and coverage of relevant studies. The validation using the QGS was done by constructing a set of representative studies and evaluating whether the search string successfully retrieved them. Details about the validation using QGS and the other review activities are documented in the protocol available in the replication package[4]. The studies used for QGS validation are listed in the file *MasterFile_LLMs_SA_SASS_Analysis.xlsx* in the worksheet *Quasi_Gold_Standard_Articles.*

The validated search string was then used for automatic search across three major digital libraries, namely ACM Digital Library, IEEE Xplore, and Scopus. The search targeted title and abstract fields and looked for studies published from January 1, 2018, to June 25, 2026[5].

[3] From this point onward, we use the term LLM to refer to both Large Language Models and Gen AI, unless specified otherwise.

[4] Replication Package: https://tinyurl.com/6jr3cd2x

[5] It is the time when Gen AI and modern LLMs started emerging.

LISTING 1: SEARCH STRING

("large language model" OR "large language models" OR LLM OR LLMs OR "generative AI" OR "generative Artificial Intelligence" OR "Gen AI" OR "Gen Artificial Intelligence")
AND
("software architecture" OR "architecture design" OR "architectural design" OR "architecture modeling" OR "architectural modeling" OR "architecture evaluation" OR "architecture analysis" OR "architectural reasoning" OR "architecture decision-making" OR "design rationale" OR "decision support" OR "architecture framework" OR "architecture pattern" OR "architecture tactic" OR "trade-off analysis" OR "quality attribute analysis" OR "design decision")
AND
(adapt* OR self* OR autonom*)

After the automatic search, the retrieved studies were screened manually with the inclusion and exclusion criteria (Table I) based on the title and abstract; if necessary, the full text was then screened. Then backward and forward snowballing was used to find other relevant studies in the screening. This process led to a final set of primary studies (Table IV) that was then systematically extracted and analysed as described below.

TABLE I: INCLUSION AND EXCLUSION CRITERIA (IC/EC)

Criteria	Description
IC1	The paper targets the use of LLMs for architectural design support for SASS.
Rationale:	This is the primary objective of the study as specified in Section I.
EC1	Non-peer-reviewed studies.
Rationale:	Non-peer-reviewed studies were excluded to ensure scientific rigor and reliability.
EC2	Non-English publications.
Rationale:	English is the primary language used by the authors.
EC3	Papers shorter than 5 pages.
Rationale:	Papers shorter than five pages were excluded because they often lack the architectural, methodological, and evaluation details required for reliable extraction of evidence addressing the review of questions.

B. Data Extraction and Analysis Method

After identifying the primary studies, both authors independently reviewed the full texts of all the included primary studies. To avoid inconsistencies in the data extraction process, a data extraction form comprising 18 data items was developed and used. Table II provides an overview of these data items, their purposes, and their mappings to the RQs. D1–D4 extract bibliographic information. D5 captures the quality score used to assess the reporting quality and rigor of the primary studies and to inform the interpretation and synthesis of the findings. It is measured using six quality items (Q1–Q6), listed in Table III.

For each primary study, each quality item was rated on a three-point scale {0, 1, 2}; 0 if it is a complete miss, 1 if it is partially satisfied, and 2 if it is completely satisfied. For instance, for the quality item Q1, a study receives a score of 0 if there is no problem definition, 1 if the problem definition is present but poorly specified, and 2 if there is an explicit problem definition. With a max score of 2 for each of the 6 quality items, the total quality score for each study thus ranges from 0 to 12.

TABLE II: DATA EXTRACTION ITEMS

ID	Item	Purpose (RQ)
D1–D4	Authors, year, title, venue	Documentation
D5	Quality score (Q1–Q6)	Quality
D6	LLM model used	RQ1
D7	Architectural activity supported	RQ1
D8	SASS lifecycle phase supported	RQ1
D9	Application domain	RQ1
D10	Role of LLM in architecture	RQ1
D11	Optimization technique	RQ2
D12	Evaluation method	RQ2
D13	Dataset / case study	RQ2
D14	Performance metrics	RQ2
D15	Reported limitations	RQ2
D16	Emerging trends	RQ3
D17	Research gaps	RQ3
D18	Future work	RQ3

The extracted data were then organized and analyzed using a combination of descriptive and qualitative content analysis techniques. Descriptive analysis was used to summarize study characteristics and identify patterns, while thematic analysis was employed to identify, compare, and group recurring concepts, themes, and findings relevant to the RQs. The data were analyzed qualitatively deductively with a coding frame that was a priori determined based on the RQ categories and the items to be extracted from the data (Table II): RQ1 was coded for D6-D10, RQ2 was coded for D11-D15, and RQ3 was coded for D16-D18. Rigor and individual coder bias were reduced by both authors independently coding all four major studies in this shared frame, then comparing assignment item by item and reaching consensus on the assignments.

TABLE III: QUALITY ITEMS USED TO ASSESS QUALITY OF THE REVIEWED STUDIES

Q1: Problem Definition	
No problem definition	0
General problem definition	1
Explicit problem definition	2
Q2: Problem Context	
No context description	0
General context description	1

Explicit context description	2
Q3: Research Design	
No description of how research was done	0
General description of how research was done	1
Explicit description of how research was done	2
Q4: Contributions	
No contributions description	0
General contributions description	1
Explicit contributions description	2
Q5: Insights	
No insights description	0
General insights description	1
Explicit insights description	2
Q6: Limitations	
No limitations description	0
General limitations description	1
Explicit limitations description	2

The coded data and supporting excerpts are available in the replication package, in the *Data_Extraction_Items* worksheet in the MasterFile_LLMs_SA_SASS_Analysis.xlsx file included in the replication package. After independently coding and analyzing the extracted data, both authors jointly reviewed their evidence, consolidated similar findings, and developed higher-level categories and themes. These themes were subsequently mapped to the RQs to provide a coherent and evidence-based synthesis of the literature. Throughout the process, an audit trail of extraction decisions, coding activities, and synthesis outcomes was maintained (see the replication package) to ensure transparency, consistency, and reproducibility of the review results.

III. Results

This section presents the outcomes of the search, study selection, and quality assessment, providing an overview of the available evidence on LLM-supported architectural design in SASS. The interpretation and analysis of these results are presented in Section IV.

A. Search and Selection Results

The automatic search yielded 1,183 studies (Scopus: 928, ACM DL: 47, and IEEE Xplore: 208). Applying filters to exclude non-English publications and items not published as peer-reviewed journal or conference articles reduced the set to 463 studies (Scopus: 226, ACM DL: 45, and IEEE Xplore: 192). After removal of duplicates, the dataset was further reduced to 363 studies. Subsequent screening using the inclusion and exclusion criteria (Table I) yielded only four primary studies (Table IV) that met all criteria. Neither backward nor forward snowballing identified any additional studies that met the inclusion criteria. The results of the search and selection process demonstrate that, despite the rapidly growing body of research on LLMs, only a very limited number of studies explicitly address the use of LLMs for architectural design in self-adaptive software systems.

The inclusion of only four primary studies constitutes a **major finding** of this review and reveals a substantial gap in the literature. The identified studies are solving the problem indirectly and are from different domains. In particular, only a handful of studies have explored the use of LLMs for supporting architectural design in SASS, indicating that the field is still at an early stage of development and remains largely underexplored.

TABLE IV: Selected primary studies (S1–S4) with bibliographic details.

ID	Bibliographic Details
S1 [15]	Yuchen Xia, Nasser Jazdi, and Michael Weyrich. An architecture for integrating large language models with digital twins and automation systems. In 2025 IEEE 30th International Conference ETFA, pp. 1–8, IEEE, 2025.
S2 [16]	Gianpaolo Ghiani, Emanuele Manni, and Sandro Zacchino. Improving adaptability in optimization-based decision support systems through large language models. IEEE Access, 2025.
S3 [17]	Çağatay Umut Öğdü, Kübra Arslanoğlu, and Mehmet Karaköse. An adaptive multi-agent LLM-based clinical decision support system integrating biomedical RAG and web intelligence. IEEE Access, 2025.
S4 [18]	Yangyang Zhuang, Wenjia Jiang, Jia-Yu Zhang, Ze Yang, Joey Tianyi Zhou, and Chi Zhang. Learning to be a doctor: Searching for effective medical agent architectures. Proc. 33rd ACM International Conference on Multimedia, pp. 6996–7005, 2025.

B. Quality Assessment Results

Before proceeding with the data extraction, analysis, and synthesis of the final four primary studies (Table IV), each study was subjected to a quality assessment to evaluate its methodological rigor and reporting quality. The assessment was conducted systematically by both authors, who independently evaluated each study against the predefined quality criteria and resolved any disagreements through discussion and consensus. The results of the quality assessment of each primary study are shown in Table V.

TABLE V: Quality Assessment Scores for Primary Studies.

Study	Q1	Q2	Q3	Q4	Q5	Q6	Total
S1	2	2	1	2	2	1	10/12
S2	2	2	2	2	2	2	12/12
S3	2	2	2	2	2	2	12/12
S4	2	2	2	2	2	0	10/12

Two studies (S2 and S3) reported at a high level (12) and the other two studies scored 10, suggesting good reporting quality. This implies that, although there are only a small number of studies of primary sources, the studies included are generally well reported. The scores should be interpreted as a sufficiently transparent basis for data extraction and analysis from the quality items, not as an assurance of methodological soundness since the quality items are more focused on reporting completeness. Each of the primary studies is assessed in detail, and justification of quality assessment scores is reported in *Reasons_for_Assigned_Quality_Scores.pdf*, which is part of the replication package, for transparency and reproducibility.

IV. Analysis And Discussion

This section analyzes and synthesizes the findings from the four primary studies (S1–S4). Given the limited number of studies, the focus is on identifying emerging patterns, architectural implications, and research directions rather than generalizable claims.

Inclusion Rationale: Since none of the primary studies use the term SASS explicitly, we document why each satisfies IC1. (S1) targets industrial automation systems that must adapt task execution at runtime: the LLM–digital-twin architecture supports replanning when exception events or execution failures are detected, making self-adaptation an explicit architectural capability. Its architectural design activity is twofold: the study contributes a three-layer reference architecture and three information-modeling design paradigms (design-time architectural design), while at runtime the LLM performs context interpretation and plan revision within that architecture, acting as a reasoning component inside a deliberately designed adaptive architecture rather than as a generic application service.

(S2) addresses an optimization-based decision support system whose hard-coded algorithms cannot follow evolving requirements; an LLM regenerate and verifies the code of decision-metric components and re-ranks candidate plans under new natural-language trade-off policies. This constitutes requirement-driven dynamic reconfiguration of system components, a canonical self-adaptation mechanism, in which the LLM directly performs the component redesign activity rather than application-level reasoning.

(S3) is an adaptive multi-agent clinical decision support system that monitors the confidence of its own outputs and, below a threshold, reconfigures its retrieval and query parameters through a bounded feedback loop, i.e., a MAPE-like control loop over its own reasoning pipeline (self-optimization). We note transparently that the LLM reasoning itself is largely application-level (diagnosis); the study is included because its contribution is the adaptive multi-agent architecture and the architecture-level design decisions it reports (agent decomposition, per-agent model selection, and loop and threshold design), making it a borderline but defensible case under IC1.

(S4) is the clearest instance of LLM-supported architectural design: the LLM analyzes diagnostic failures and applies node-, structural-, and framework-level modifications to the agent's workflow, performing automated architecture search and evolution. The managed element is the agent architecture itself, which adapts autonomously architectural self-adaptation in its strongest form, albeit directed at the agent system rather than an external operational process.

A. RQ1: Use of LLMs for Architectural Design Support in SASS

The four reviewed studies do not use LLMs as first-class architectural elements, but rather as augmentative reasoning components that operate alongside existing system components to assist in interpreting context, generating and refining design alternatives, supporting trade-off analysis, and providing explanations for architectural decisions. There is an obvious separation between design-time and run-time use. (S2) combines design-time reconfiguration with runtime decision support: the LLM generates and self-verifies decision-metric code when company policies change (design time) and selects among candidate plans according to natural-language trade-off preferences each time a new request arrives (runtime), thereby adapting an optimization-based decision support system without redesigning it. (S1) and (S3), in turn, operate primarily at runtime: in (S1), LLM agents interpret context from digital twins and produce control plans for industrial automation, while in (S3), a multi-agent clinical decision support system interprets patient context, retrieves biomedical evidence, and makes diagnostic decisions through a confidence-driven feedback loop.

(S4) is distinctive in that the LLM operates on the agent architecture itself: framed as an automated, AutoML-inspired search over multi-agent diagnostic workflows, it iteratively adds, removes, and modifies workflow nodes and control structures in response to diagnostic feedback. From an architectural perspective for SASS, the reviewed studies show that LLMs primarily support the monitoring, analysis, planning, and knowledge phases of the MAPE-K loop [5], contributing to context understanding, reasoning, and decision support, with execution-phase involvement appearing only partially, in (S4), where the LLM enacts changes to the agent architecture rather than to a managed operational system. All four studies rely on an explicit or implicit knowledge component, whether a digital twin (S1), an optimization model (S2), a biomedical retrieval store (S3), or a structured search space over agent workflows (S4).

Together, these observations indicate that LLMs in the reviewed studies primarily support architectural reasoning rather than being incorporated as explicit components within the architectural design of SASS. The MAPE-K coverage across the four studies is summarized in Table VI, and the justification of inclusion is given in Table VII. The coding rules for a study to be mapped to a MAPE-K phase were a study explicitly described as the functionality of LLM that fulfilled the rules of the phase.

Monitor: The LLM (or part of its chain of execution) repeatedly receives information at runtime about the managed system or its environment; A single user prompt is not sufficient.

TABLE VI: MAPE–K Phase Coverage Across the Identified Primary Studies

Study	Monitor	Analyze	Plan	Execute	Knowledge
S1	✓	✓	✓	×	✓
S2	×	✓	×	×	✓
S3	✓	✓	×	×	✓
S4	✓	✓	✓	partial	✓

Analyze: monitor context for deviation, diagnose cause, evaluate alternatives based on stated preferences, etc. producing a recommendation is not planning.

Plan: creation of an actionable adaptation plan or change set for the managed system, for example, control plans or workflow changes but not refinement of own queries or analyses.

Execute: The adaptation is implemented without the intervention of a human being; if the adaptation is implemented with a human intervention, then the enactment is coded as no execution support.

Knowledge: explicit shared knowledge components (models, indexes, repositories, etc.) consumed by phases of a loop.

With these rules, the outputs of S1 pass through human validation, and the outputs of S3 are handled by the system's own adaptive optimizer, which modifies the system's own queries, but S4 is written as partial Execute because validated modifications are applied automatically, but do not affect an external managed system they affect the agent architecture that is being managed. Evidence excerpts supporting each cell of Table VI are provided in the replication package.

B. RQ2: Optimization and Evaluation Techniques

The reviewed studies employ a range of techniques to improve LLM performance and reliability, including structured prompting and prompt templating, retrieval-augmented grounding, multi-agent decomposition, comprehension verification with self-debugging (S2), confidence-driven feedback loops (S3), and automated search over agent architectures (S4). These aim to mitigate well-known LLM limitations such as hallucination and lack of grounding; (S1) additionally relies on human validation and test-driven development, while (S3) constrains hallucination through biomedical retrieval and cross-checking against web evidence. From an architectural perspective, however, these techniques remain largely external to the software architecture, not explicitly integrated into architectural models or frameworks. Evaluation approaches are heterogeneous and domain specific, focusing on task performance rather than architectural quality attributes such as robustness, scalability, or adaptability.

(S3) and (S4) report task accuracy on domain benchmarks (medical question answering and dermatological image diagnosis, respectively), (S2) evaluates correctness, robustness to ill-defined prompts, and interpretability on a vehicle-routing case, and (S1) provides qualitative proof-of-concept case studies rather than benchmark metrics. Notably, even (S4), whose explicit goal is architecture search, evaluates the task accuracy, output consistency, and resource cost of the resulting workflows, but no architectural qualities such as adaptability or maintainability. The absence of architecture-aware evaluation limits comparability across studies and indicates a need for benchmarks and frameworks tailored to LLM-supported architectural design for SASS.

C. RQ3: Recurring Characteristics and Research Gaps

Despite the limited number of studies, several commonalities can be observed. Given the small and heterogeneous sample, we frame these as recurring characteristics of the selected studies rather than broader trends in the field, as it is difficult to distinguish a meaningful trend from an incidental property of one or two studies. The observed characteristics include the adoption of hybrid architectures based on using the LLM in conjunction with other components, such as digital twins (S1) and optimization engines (S2); the increased use of multi-agent reasoning, which is being automated in (S4) by searching for and evolving agent architectures; and the adoption of grounding mechanisms, such as retrieval augmentation and confidence-based feedback, to ensure reliability (S3). Meanwhile, there are still many areas of research that need to be addressed.

The first is the absence of architecture focused integration frameworks that treat LLM as explicit architectural components in SASS instead of as add-ons. Second, trustworthiness, explainability and assurance issues are not adequately investigated, either in the case of (S1), where the work is based on human validation; or in the case of (S3), where the level of confidence is set in a specific way; in both, there are no guarantees in the architecture. Third, there is no support for variability management, which is at the core of the SASS and their product lines. Last, but not least, there is a lack of a standardized and architecture-aware evaluation approach that hinders comparison and generalization. The identified gaps suggest that the research into using LLMs for architectural design in SASS is still in its exploratory stages, and there is a significant amount of work still to be done in order to create robust and architecture-centric approaches.

TABLE VII: CROSS-STUDY SYNTHESIS OF ARCHITECTURAL ROLES AND ADAPTIVE CHARACTERISTICS.

Study	Architecture	LLM Role	Adaptation Mechanism
S1 [15]	Layered Digital-Twin (DT)	Coordinator	DT-mediated adaptation; feedback; runtime
S2 [16]	Hybrid	Decision reasoner	Dynamic reconfiguration; runtime
S3 [17]	Multi-agent	Orchestrator	Feedback-driven optimization
S4 [18]	Graph-based	Architecture designer	Workflow evolution; iterative feedback

D. Synthesis and Implications

In the reviewed studies, LLMs play a role of more than just an AI service; they are responsible for coordinating components, orchestrating reasoning processes, integrating heterogeneous knowledge sources, and supporting architectural evolution. Overall, the results suggest that LLM are currently not fully embedded in SASS frameworks with a focus on architecture but are employed to enhance the human and system reasoning processes by providing context interpretation, decision support and explanations. The almost complete lack of some execution support (only partially present in (S4), where the LLM modifies and deploys the agent architecture), as well as architecture-based integration and variability management, further highlights the limited execution support and the need to strengthen it for SASS.

The two studies closest to treating architecture as a first-class concern were (S1), with its layered LLM–digital twin automation design approach, and (S4), with its automated architecture search approach; both hint at a possible direction of travel: embedding the LLM not just as a thinking resource, but as a co-creative design agent. This indicates that there is a need for new architectural models that explicitly integrate LLMs in the MAPE-K loop and tackle important issues like trustworthiness, adaptability, and scalability. The results must be viewed as preliminary insights into possible future research avenues and should serve as a starting point for more systematic, architecture-informed use of LLMs in SASS, given the limited amount of evidence.

V. Threats And Validity

This section outlines the main threats to the validity of the study, the measures taken to mitigate them, and the residual threats that remain despite these measures.

- **Search validity** concerns whether or not relevant studies could be obtained. The RQs were used to develop the search terms, which were narrowed down through iterative search and verified using the QGS method, and backward and forward snowballing was adopted to enhance the search coverage. Even with these efforts, there is a possibility of the residual threat of terminology bias: no terminology is yet standardized, and other related studies, such as studies of self-adaptive systems that are referred to by different terms (e.g., autonomic computing, agentic AI, dynamic reconfiguration, or specific self-* properties), or studies that do not use self-adaptive in the title or abstract, might not have been retrieved. The wide-ranging wildcard terms (adapt*, self*, autonom*) in the search string diminish, but do not completely remove, this risk.
- **Selection validity** addresses bias in the selection of studies (inclusion/exclusion) that is addressed by having two-step screening with pre-established inclusion/exclusion criteria with disagreements resolved by consensus. There remains a subjective issue about the relevance of the studies for architectural design support for SASS (IC1): When evaluating the relevance of studies for architectural design support for SASS, decisions are needed about these studies and whether they explicitly use the SASS terminology or address the issue of adaptivity only indirectly. Although independent screening and consensus minimize individual interpretation bias, other researchers may have interpreted the same borderline studies differently. In particular, S3 was the most contested inclusion decision; we retained it because its contribution is the adaptive multi-agent architecture rather than the diagnostic task itself, but other reviewers could reasonably have excluded it.
- **Data extraction and quality assessment validity** concerns the accuracy and interpretation of data extracted and quality scores; addressed by use of pre-established data extraction items that were related to the RQs, each author extracting and assessing independently, and discrepancies resolved by discussion. Even so, qualitative coding and quality scoring remain interpretive activities, and residual subjectivity in mapping four heterogeneous studies to a common coding frame cannot be fully excluded.
- **External validity** is a question of generalizability. The results are considered preliminary indications only, because there are only four primary studies, which reflects the **newness of the area**. This threat cannot be mitigated at this stage; instead, we address it by framing the findings as recurring characteristics of the selected studies rather than generalizable trends.

Further details are provided in the protocol document in the replication package.

VI. Future Works

Given the limited number of primary studies, we plan to extend the scope of the literature review as future work. Specifically, we aim to review studies from our search results that investigate the use of LLMs for software architectural design in general, rather than being limited to SASS. Using this extended scope, we have already screened the search results and identified 58 additional studies. These studies will be examined in future work to derive insights relevant to SASS. A complete list of these studies is given as a worksheet *LLMs_and_SoftwareArchitecture* in the file *MasterFile_LLMs_SA_SASS_Analysis.xlsx* available in the replication package. As a next step, we plan to analyze these studies to determine which LLM-based architectural design techniques can be transferred to or adapted for SASS, and to identify SASS-specific concerns, such as runtime adaptation, variability management, and assurance requirements, that may not be adequately addressed by approaches for general software architecture design.

A. Practical Implications

The results indicate that software architects could leverage LLMs for decision-making tasks during architectural analysis, alternative evaluation, and architectural documentation of SASS. Current LLM-based systems seem to perform best when they are used alongside expert reasoning, such as engaging in a process of exploratory design space exploration. To guarantee trustworthy and safe architectural choices, it is crucial that organizations implement mechanisms for verification, traceability, and human oversight when incorporating LLM into their architectural workflows.

VII. Conclusion

This paper shares insights from a work-in-progress systematic review on the use of LLMs in support of architectural design in SASS. The findings show that this research area is still in its infancy: among the four primary studies identified, the central observation is that the topic is not yet widely researched. So far, LLMs are used as augmentative reasoning components that support the interpretation of context, decision-making, and architectural rationale across the monitoring, analysis, planning,

and knowledge phases of the MAPE-K loop, with execution-phase involvement appearing only partially in a single study (S4). Their use as explicit architectural elements remains limited. Although the evidence base is small, the study contributes an initial characterization of LLM-supported architectural design in SASS, a first look at the different roles of LLMs across the MAPE-K loop, and an identification of key research gaps, namely architecture centric integration, standardized evaluation, and support for variability and runtime assurance. As future work, we plan to analyze the additional 58 studies on software architecture design beyond SASS to learn from the LLM-based architectural design practices, techniques, and methods they report. The goal is to identify approaches that can be transferred to or adapted for SASS and to support architects in addressing the complexity of SASS architectural design.

REFERENCES

[1] I. Mistrík, R. M. Soley, N. Ali, J. Grundy, and B. Tekinerdogan, Software quality assurance: in large scale and complex software-intensive systems. Morgan Kaufmann, 2015.

[2] W. Hasselbring, Software Architecture: Past, Present, Future. Cham: Springer International Publishing, 2018, pp. 169–184. [Online]. Available: https://doi.org/10.1007/978-3-319-73897-0_10

[3] N. Abbas and J. Andersson, "Architectural reasoning for dynamic software product lines," in Proceedings of the 17th International Software Product Line Conference Co-located Workshops, 2013, pp. 117–124.

[4] N. Abbas and J. Andersson, "Architectural reasoning support for product lines of self-adaptive software systems-a case study," in European Conference on Software Architecture. Springer, 2015, pp. 20–36.

[5] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," ACM transactions on autonomous and adaptive systems (TAAS), vol. 4, no. 2, pp. 1–42, 2009.

[6] D. Weyns, I. Gerostathopoulos, N. Abbas, J. Andersson, S. Biffl, P. Brada,T. Bures, A. Di Salle, M. Galster, P. Lago et al., "Self-adaptation in industry: A survey," ACM Transactions on Autonomous and Adaptive Systems, vol. 18, no. 2, pp. 1–44, 2023.

[7] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," ACM Transactions on Intelligent Systems and Technology, vol. 16, no. 5, pp. 1–72, 2025.

[8] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," ACM Transactions on Software Engineering and Methodology, vol. 33, no. 8, pp. 1–79, 2024.

[9] A. S. Shethiya, "Llm-powered architectures: Designing the next generation of intelligent software systems," Academia Nexus Journal, vol. 2, no. 1, 2023.

[10] J. Li, M. Zhang, N. Li, D. Weyns, Z. Jin, and K. Tei, "Exploring the potential of large language models in self-adaptive systems," in Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2024, pp. 77–83.

[11] R. Donakanti, P. Jain, S. Kulkarni, and K. Vaidhyanathan, "Reimagining self-adaptation in the age of large language models," in 2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C). IEEE, 2024, pp. 171–174.

[12] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering– a systematic literature review," Information and software technology, vol. 51, no. 1, pp. 7–15, 2009.

[13] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, ser. EASE. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2601248.2601268

[14] H. Zhang and M. Ali Babar, "On searching relevant studies in software engineering," 2010.

[15] Y. Xia, N. Jazdi, and M. Weyrich, "An architecture for integrating large language models with digital twins and automation systems," in 2025 IEEE 30th International Conference on Emerging Technologies and Factory Automation (ETFA). IEEE, 2025, pp. 1–8.

[16] G. Ghiani, E. Manni, and S. Zacchino, "Improving adaptability in optimization-based decision support systems through large language models," IEEE Access, 2025.

[17] Ç. U. Ögdü, K. Arslano ˘ glu, and M. Karaköse, "An adaptive multi-agent llm-based clinical decision support system integrating biomedical rag and web intelligence," IEEE Access, 2025.

[18] Y. Zhuang, W. Jiang, J.-Y. Zhang, Z. Yang, J. T. Zhou, and C. Zhang, "Learning to be a doctor: Searching for effective medical agent architectures," in Proceedings of the 33rd ACM International Conference on Multimedia, 2025, pp. 6996–7005.

Authors Index

Abbas, Nadeem, *106*
Antônio Temochko Andre, João, *33*
Arcelli Fontana, Francesca, *73*

Bein, Tobias, *98*
Beletti Ferreira, Alexandre, *33*
Ber, Jan, *46*
Bhatnagar, Avik, *1*
Biffl, Stefan, *98*
Bluemke, Ilona, *90*
Bringmann, Oliver, *1*
Buccellato, Federico, *65*

Can, Ahmet Zahit, *17*
Chrysos, Grigorios, *57*
Claudepierre, Ludovic, *9*

De Sio, Corrado, *65*
Dippold, Matthias, *57*
Drzensla, Łukasz, *40*

Ferrucho-Alvarez, Edna Rocio, *9*
Fijałek, Norbert, *90*

Gawrysiak, Piotr, *90*

Ioannidis, Sotiris, *57*

Koryciak, Sebastian, *40*
Kropatschek, Sebastian, *98*

Le Brizoual, Laurent, *9*
Lüder, Arndt, *98*

Machado da Silva, José, *51*
Maia Aguiar, João, *51*
Mannini, Luca, *65*
Maragkou, Sofia, *57*
Meixner, Kristof, *98*

Pacalet, Renaud, *25*
Paisi, Giorgia, *73*
Peccia, Federico Nicolás, *1*
Peczyński, Vadim, *80*
Pichon, Laurent, *9*

Russek, Paweł, *40*

Sauter, Thilo, *57*
Sauvage, Laurent, *25*
Savchenko, Olksandr, *40*
Shahzadi, Nazia, *106*
Szłapczyńska, Joanna, *80*

Uslu, Erkan, *17*

Walter, Bartosz, *73*
Wess, Matthias, *57*
Wintersdorff, Raphaël, *25*

Supported by:

ISSN 2980-7298