DSD'2024 and SEAA'2024 Works in Progress Session AUG 2024

Many a Little Makes a Mickle: On Micro-Optimisation of Containerised Microservices

Zheng Li
School of Electronics, Electrical Engineering and Computer Science
Queen's University Belfast
Belfast, United Kingdom
ORCID: 0000-0002-9704-7651

Abstract— **Performance optimisation is a key to the success of microservices architecture. Correspondingly, many studies have been conducted on optimising orchestration or composition of multiple microservices within different application contexts. Unlike the existing efforts on the global optimisation, we are concerned with the internal optimisation of individual microservices. Considering the loosely coupled nature of individual microservices, their performance improvements could be independent of each other and thus would naturally bring benefits to their composite applications. Driven by such intuitive ideas together with the de facto tech stack, we have been working on micro-optimisation of containerised microservices at the Operation side (i.e., Ops-side optimisation) against the Development side. Based on both theoretical discussions and empirical investigations, our most recent work delivered three micro-optimisation principles, namely *just-enough containerisation*, *just-for-me configuration*, and *just-in-time compilation (during containerisation)*. Our current research outcomes have not only offered new ideas and practical strategies for optimising microservices, but they have also expanded the conceptual scope and the research field of software micro-optimisation.**

Keywords-containerisation; DevOps; micro-optimisation; microservice; Ops-side optimisation; performance engineering

I. Introduction

Since microservice-based applications may suffer from intrinsic performance penalties due to their distributed nature [1], it has been identified that performance optimisation is a key to the success of microservices architecture [2]. Unlike the existing studies that directly aim at application-level optimisation (e.g., through resource provisioning [3] or microservice placement [4]), we wonder if there are opportunities to focus on every single microservice to optimise. Intuitively, and depending on the application topology, a many-microservice application may obtain enormous benefits if every microservice contributes a little performance improvement. Even if the eventual application benefits are only "small efficiencies" [5], the microservice-level optimisation will still be worth it, because modern (Web) applications are generally performance sensitive, e.g., "every drop of 20 ms... latency will result in a 7–15% decrease in page load times" [6].

Driven by these intuitive ideas, we defined the following research question to unfold concrete investigations and to distinguish from the application-level optimisation studies:

RQ: Can we, and if yes, how do we conduct application-agnostic optimisations to improve the performance of individual microservices?

Inspired by the single-purpose feature of microservices, our current investigation efforts are paid to the micro-optimisation opportunities. Traditionally, low-level optimisation may not be considered as a good idea because it tends to be platform-centric [7]. When it comes to the containerised microservices, the containerisation mechanism encapsulates microservice components and their runtime environment all together, which means that the microservices' platform details (e.g., the OS kernel, word size, and CPU instruction set architecture) are all settled in advance. Then, there will be nothing wrong to conduct platform-centric optimisations in this situation.

Moreover, given the convergence of infrastructure as code and container technologies [8], containerised microservices should be able to enjoy more micro-optimisation opportunities, not only at the Development side (i.e., Dev-side optimisation in source code files) but also at the Operation side (i.e., Ops-side optimisation in Dockerfile, shell scripts, and machine-readable definition files). To verify this new idea, we further narrow down our focus to micro-optimisation with respect to containerisation. At the time of writing, our ongoing work has developed three micro-optimisation principles, and we name them as *just-enough containerisation*, *just-for-me configuration*, and *just-in-time compilation (during containerisation)*. By reporting and justifying these principles, this paper makes a twofold contribution:

- In theory, this work expands the conceptual scope of, and reveals new research opportunities in the field of, software micro-optimisation. To our best knowledge, this is the first study that advocates and investigates the low-level, Ops-side optimisation.
- In practice, this work suggests new strategies to optimise containerised microservices. These strategies would be increasingly applicable, along with the fast-growing DevOps ecosystem that heavily leverages infrastructure as code and container technologies.

II. Related Work

A. Microservices Optimisation

Exploring "microservice(s) optimisation" in the literature will bring a tremendous number of studies on optimising orchestration or composition of multiple microservices, within different application contexts. By distinguishing between scalable (e.g., cloud) and restricted (e.g., user device) runtime environments, the existing studies generally formulate microservices optimisation either as resource-provisioning problems or as microservice-placement problems, with different objectives and different solution proposals.

In particular, we have observed a broad range of optimisation objectives including, for example, minimising response time/latency [4], resource consumption/cost [3], [9], energy consumption [10], failure rate [10], etc.; and maximising reliability [4], [9], resource utilisation efficiency [3], network throughput [10], load balancing [4], [9], etc. Correspondingly, the proposed solutions also vary hugely, such as particle swarm optimisation, non-dominated sorting genetic algorithm III (NSGA-III), fine-tuned sunflower whale optimisation algorithm, ant colony algorithm, knowledge-driven evolutionary algorithm, Lagrangian multipliers, etc.

It is worth noting that although some researchers claim to have worked on the performance tuning [11] and configuration adjustment [12] of individual microservices, their research work still measures and refers to the application indicators to conduct the optimisation. In contrast, our research focuses on the application-agnostic optimisation of microservices; and more distinctively, we aim to optimise microservices during their containerisation process before the runtime execution.

B. *Software Micro-Optimisation*

Since we have not found any study on micro-optimisation of microservices, we consider the generic software micro-optimisation as a related topic to our research.

Software micro-optimisation is generally defined as the source code-level optimisation (e.g., using StringBuilder instead of String in Java) without changing the software architecture, design, and algorithms [13]. According to the literature, the community seems to have opposite opinions about software micro-optimisation. By citing Sir Tony Hoare's famous quote "premature optimisation is the root of all evil" (popularised by Donald Knuth) [5], "we should forget about small efficiencies" has been argued as a best practice of software engineering and even as a rule of programming [7]. Except for the unawareness of software micro-optimisation by some practitioners, the main concern is that micro-optimisation may not be a worthwhile investment of time compared to macro-optimisation [13], [14].

In the meantime, there are also advocates of software micro-optimisation. A direct response to the aforementioned concern is that it is always worth investing developers' time to save software users' time [5]. Another investigation empirically justifies how worthwhile the micro-optimisation can be, by tweaking a piece of code that is responsible for a substantial proportion of the execution time [14]. After all, given the software crisis behind the continuously growing CPU power, it is never enough to emphasise the optimisation of software systems.

We are also convinced of the value of software micro-optimisation, because "any (even small) performance improvement will matter" in modern computing paradigms (e.g., IoT, edge, fog, cloud) [6]. Particularly, we are further concerned with the micro-optimisation at the Ops side instead of Dev side.

III. Three Micro-Optimisation Principles

To better introduce the three micro-optimisation principles, we particularly highlight the justification for each principle description, in separate subsections.

A. *Just-enough Containerisation*

1) Principle Description: When wrapping up target functionalities into a containerised microservice, the containerisation should include just-enough software components and minimise the installation of just-in-case programs and middleware.

2) Justification: When it comes to deploying a microservice-based application, it has been widely discussed that the co-located microservices in a multi-tenant environment can interfere with and slow down each other due to the competition for non-partitionable resources, and the resource competition may eventually cause unpredictable behaviour and performance degradation of the microservice-based application [15], [16].

In fact, if zooming into an individual microservice, we can also expect to see the resource competition among its co-located components. Recall that each microservice is a (single-purpose) software application and controls its own data [17]. Since a single-purpose application is still composed of multiple components (e.g., codebase modules and a database management system), a containerised microservice can include multiple containers, and each container encapsulates a software component together with the component's entire runtime environment. Therefore, such a multi-container microservice should naturally be recognised as a multi-tenant system.

Furthermore, to support the main functionalities, each containerised software component may also install its enabling services, registry entries, background tasks, drivers, shared libraries, etc. on the fly during the image building process. Thus, the containerisation of any software component would incur extra performance overhead. Even if there is no resource competition by those inactive software components at runtime, the unneeded installations will unnecessarily increase the size of the corresponding microservice, and it has been empirically identified that the unused stuff is the major cause of memory waste [13].

B. *Just-for-me Configuration*

1) Principle Description: Without changing the codebase and the predefined tech stack, it is worth customising the configurations of microservice components during the containerisation, to better support the microservice's non-functional features.

2) Justification: Aligning with the single-responsibility principle "do one thing and do it well", the loosely coupled microservices are supposed to work (largely) independently on different single purposes, even within the same application context. Since each microservice may have its own and unique runtime characteristics, there does not exist a one-size-fits-all configuration to maximise the potentials of different microservices. For example, without understanding the data needs of individual microservices and accordingly planning respective strategies, the efficient cache configurations for one microservice (e.g., Django's caching framework and MySQL query cache) could result in inefficient caching and performance degradation for another microservice [18].

Another typical example is when containerising microservices that involve database systems, a microservice can achieve higher performance by optimising its database performance against its specific workload and dataset [19]. For instance, we can create additional indexes to expedite data retrieval for stateless queries. Besides the existing database tuning tips, in this paper we particularly report our experience in tweaking read-only database containers to exemplify and justify the micro-optimisation's effectiveness and efficiency.

According to Docker, the official mechanism of containerising a read-only database is to use the read-only parameter to specify a mounted data volume as read-only.[1] However, mounting data volumes "by punching a hole through the container" has been considered unreliable [20], and thus it has been argued that containers cannot be a secure candidate solution for database [21]. In our project, we advocate pre-baking read-only data into container images to avoid mounting external volumes. By making such a customisation, this unofficial mechanism can not only intrinsically enable read-only databases (because the pre-baked data are immutable in image) but also improve the reliability of database containers (because no "hole" exists through the container).

We have also compared the data retrieval performance between these two mechanisms of containerising read-only databases. The testbed was set up on a clean HP OMEN laptop of model 17-an003la (with Intel's four-core CPU Core™ i7-7700HQ at 2.8 GHz base frequency). After re-installing 64-bit Ubuntu 20.04 as the operating system (OS), we further installed Docker 20.10.2, Python 3.8.5, and Jupyter Notebook 3.8.5. To facilitate observation, we intentionally employed large-size datasets (up to about 160 MB) to measure the latency of data retrieval from MySQL 5.7.34, in order to magnify the performance difference.

Given the experimental results[2] as shown in Figure 1, it is clear that pre-baking data into image can even achieve performance advantage over mounting data volumes. In other words, this micro-optimisation can bring multiple non-functional benefits to read-only database containers as well as their supported microservices

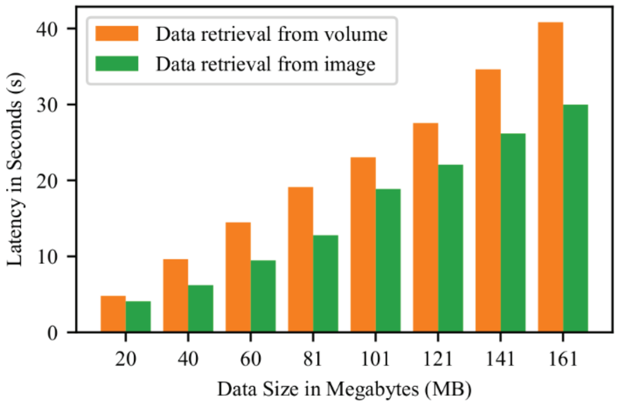

Figure 1. Data retrieval performance comparison between two mechanisms of read-only database containerisation.

C. Just-in-time Compilation (during Containerisation)

1) Principle Description: When applicable, it is worth compiling the interpreted microservice components during the containerisation process, to avoid (or at least minimise) the interpretation overhead in the runtime of containerised microservices.

2) Justification: Driven by the needs of reducing development costs cross heterogeneous platforms, "write once, run anywhere" has become a standard practice in software industry. Despite various implementations of this standard practice, the essential enabling technique is to equip the development with a pervasively installed middleware (e.g., a runtime framework or a code interpreter) that abstracts the underlying details of different platforms. Nevertheless, constrained by the No-Free-Lunch theorem of optimisation [22], it is also well known that the middleware-aided code translation will impose a performance penalty whenever it runs.

To alleviate the performance penalty, there emerges a runtime compilation strategy, i.e., translating source code or bytecode into machine code during the first-time execution of a program. However, this will meanwhile introduce a warm-up latency to the executables due to the extra computational overhead for interpreting, compiling, and linking the code.

Recall that one of the best practices of containerisation is to not only import an OS base image but also specify the OS version [23], while the container images must have targeted one or more specific platforms (processor architectures) [24]. In other words, a container's production environment is already (pre-)fixed when preparing its image. Therefore, we shall be able to compile the to-be-containerised microservice components just in time when building container images. In this way, we can further avoid the aforementioned warm-up latency of the relevant microservice components, and ultimately improve the overall microservice performance at runtime.

In addition to the codebase components, the database part of a microservice also has similar micro-optimisation opportunities. Conventionally, given a SQL query, the database system will first convert the query into an execution plan (i.e., a sequence of data access steps) and then execute the plan via interpretation. Since the performance of modern query engines is increasingly dominated by the memory access and CPU usage, there is an emerging trend in dropping interpretation in favour of compilation [25]. As demonstrated by a quantitative study on compiling a set of selected benchmark queries [26], although the performance advantage varies case by case, the query execution after compilation generally outperforms the query execution via interpretation.

Similarly, the execution performance advantage also comes with an extra overhead of compilation, which may make the overall query processing take even longer time. To address this problem, unlike the current research efforts that dominantly aim

[1] https://docs.docker.com/storage/volumes/
[2] The experimental source files and documentation are shared at https://doi.org/10.5281/zenodo.8341856

to expedite runtime and session-specific compilation of dynamic user queries [27], we argue to natively compile the pre-known microservice-specific queries just in time during containerisation, because this will extinguish the runtime compilation overhead when executing the containerised queries. In fact, we have seen promising techniques that are aligned with this idea, e.g., SQL Server supports native compilation of tables and stored procedures into DLLs.[3]

IV. Conclusions and Future Work

Performance optimisation is crucial and valuable for the implementation of microservices architecture. In addition to globally optimising microservice-based applications by adjusting resource provisioning and/or microservice placement, we argue that the internal micro-optimisation of individual microservices would also bring enormous benefits to microservice-based applications. Although still at an early stage, our theoretical discussions and empirical trials have led to a set of micro-optimisation principles with initial validation of their effectiveness and efficiency, at least at the Operation side for containerised microservices.

On the other hand, there are clearly needs to enrich empirical evidence for strengthening our developed principles and for proposing new ones. Therefore, this early-stage research points out two directions toward the immediate future work. Firstly, it is worth keeping trying different micro-optimisation techniques and quantitatively studying their effects in the context of a single microservice. Secondly, it will be helpful to use the third-party application benchmarks [4] to observe and investigate the combined and overall effects of micro-optimising multiple microservices.

References

[1] S. Baškarada, V. Nguyen, and A. Koronios, "Architecting microservices: Practical opportunities and challenges," *J. Comput. Inf. Syst.*, vol. 60, no. 5, pp. 428–436, 2020.

[2] Firdavs, "Quality Attribute Analysis in Microservices Architectures" https://dev.to/firdavsm1901/quality-attribute-analysis-in-microservices-architectures-33li, 13 Apr. 2024.

[3] M. Kumar, J. K. Samriya, K. Dubey, and S. S. Gill, "QoS-aware resource scheduling using whale optimization algorithm for microservice applications," *Softw.: Pract. Exper.*, vol. 54, no. 4, pp. 546–565, Apr. 2024.

[4] G. Fan, L. Chen, H. Yu, and W. Qi, "Multi-objective optimization of container-based microservice scheduling in edge computing," *Comput. Sci. Inf. Syst.*, vol. 18, no. 1, pp. 23–42, Jan. 2021.

[5] R. Hyde, "The fallacy of premature optimization," *ACM Ubiquity*, vol. 10, no. 3, Feb. 2009, art. no. 1.

[6] Z. Li and J. Galdames-Retamal, "On iot-friendly skewness monitoring for skewness-aware online edge learning," *Appl. Sci.-Basel*, vol. 11, no. 16, Aug. 2021, art. no. 7461.

[7] B. Smaalders, "Performance anti-patterns: Want your apps to run faster? here's what not to do." *ACM Queue*, vol. 4, no. 1, pp. 44–50, Feb. 2006.

[8] J. Das, "Making infrastructure as code a better framework with containers," https://blog.aspiresys.com/infrastructure-managed-services/making-infrastructure-as-code-a-better-framework-with-containers/, 28 Sept. 2022.

[9] M. Lin, J. Xi, W. Bai, and J. Wu, "Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud," *IEEE Access*, vol. 7, pp. 83 088–83 100, Jun. 2019.

[10] A. Samanta and J. Tang, "Dyme: Dynamic microservice scheduling in edge computing enabled IoT," *IEEE Internet Things J.*, vol. 7, no. 7, pp. 6164–6174, Jul. 2020.

[11] V. M. Mostofi, D. Krishnamurthy, and M. Arlitt, "Fast and efficient performance tuning of microservices," in *Proc. CLOUD 2021*. Chicago, IL, USA: IEEE Press, 05-10 Sept. 2021, pp. 515–520.

[12] H. Dinh-Tuan, K. Katsarou, and P. Herbke, "Optimizing microservices with hyperparameter optimization," in *Proc. MSN 2021*. Exeter, United Kingdom: IEEE Press, 13-15 Dec. 2021, pp. 685–686.

[13] M. Linares-Vásquez, C. Vendome, M. Tufano, and D. Poshyvanyk, "How developers micro-optimize Android apps," *J. Syst. Soft.*, vol. 130, pp. 1–23, Aug. 2017.

[14] A. Trotman and M. Crane, "Micro- and macro-optimizations of SaaT search," *Softw.: Pract. Exper.*, vol. 49, no. 5, pp. 942–950, May 2019.

[15] Y. D. Barve, S. Shekhar, A. Chhokra, S. Khare, A. Bhattacharjee, Z. Kang, H. Sun, and A. Gokhale, "FECBench: A holistic interference-aware approach for application performance modeling," in *Proc. IC2E 2019*. Prague, Czech Republic: IEEE Press, 24-27 Jun. 2019, pp. 211–221.

[16] D. Masouros, S. Xydis, and D. Soudris, "Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 184–198, Jan. 2021.

[17] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, and Z. Shan, "A dataflow-driven approach to identifying microservices from monolithic applications," *J. Syst. Softw.*, vol. 157, Nov. 2019, art. no. 110380.

[18] S. K. Shivakumar, "Web performance monitoring and infrastructure planning," in *Modern Web Performance Optimization: Methods, Tools, and Patterns to Speed Up Digital Platforms*. Berkeley, CA: Apress, Nov. 2020, ch. 7, pp. 175–212.

[19] K. Kanellis, R. Alagappan, and S. Venkataraman, "Too many knobs to tune? towards faster database tuning by pre-selecting important knobs," in *Proc. HotStorage 2020*. USENIX Association, 13-14 July 2020, art. no. 16.

[20] J. Tobin, "Are docker containers good for your database?" https://www.percona.com/blog/2016/11/16/is-docker-for-your-database/, 16 Nov. 2016.

[21] S. Shirinbab, L. Lundberg, and E. Casalicchio, "Performance evaluation of containers and virtual machines when running Cassandra workload concurrently," *Concurrency Comput. Pract. Exper.*, vol. 32, no. 17, Feb. 2020, art. no. e5693.

[22] F. Rabhi, M. Bandara, A. Namvar, and O. Demirors, "Big data analytics has little to do with analytics," in *ASSRI 2015, ASSRI 2017: Service Research and Innovation, ser.* Lect. Notes Bus. Inf. Process., A. Beheshti, M. Hashmi, H. Dong, and W. E. Zhang, Eds. Cham: Springer, Mar. 2018, vol. 234, pp. 3–17.

[23] Y. Wu, Y. Zhang, T. Wang, and H. Wang, "Characterizing the occurrence of dockerfile smells in open-source software: An empirical study," *IEEE Access*, vol. 8, pp. 34 127–34 139, Feb. 2020.

[24] A. Mouat, "Multi-platform docker builds," https://www.docker.com/blog/multi-platform-docker-builds/, Mar. 2020.

[25] T. Neumann, "Evolution of a compiling query engine," *Proc. VLDB Endow.*, vol. 14, no. 12, pp. 3207–3210, Jul. 2021.

[26] A. Rayabhari, "Compilation-based execution engine for a database," https://www.cs.cornell.edu/courses/cs6120/2020fa/blog/db-compiler/, 18 Dec. 2020.

[27] H. Funke and J. Teubner, "Low-latency compilation of SQL queries to machine code," *Proc. VLDB Endow.*, vol. 14, no. 12, pp. 2691–2694, Jul. 2021.

[3] https://learn.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/native-compilation-of-tables-and-stored-procedures?view=sql-server-ver16

[4] https://github.com/delimitrou/DeathStarBench