WASM and Containers for Real-Time Serverless Edge Computing

Isser Kadusale, Gautam Gala, and Gerhard Fohler
Chair of Real-Time Systems,
Technical University of Kaiserslautern-Landau (RPTU)
isser.kadusale@rptu.de, gala@eit.uni-kl.de, gerhard.fohler@rptu.de

***Abstract*—**
A new trend of offloading real-time (RT) systems to edge computing platforms is emerging. Function-as-a-Service, a type of serverless computing, has useful applications for RT systems by offloading aperiodic tasks to the edge, or by reducing over-provisioned periodic tasks. Due to their comparatively low overheads, containers are an attractive alternative to VMs for supporting RT serverless computing. However, containers still have relatively long cold startup times. WebAssembly (WASM) has been suggested for use in non-RT serverless edge computing. Due to its short cold startup times, it can also be a potential alternative to containers. We explore this by running synthetic benchmark tests to see the timing performance of WASM runtimes in comparison to a container. We compare the startup times and execution times of different WASM runtimes and a container, with the goal of determining which of them is better in which situations. Furthermore, we discuss the qualitative suitability of WASM as compared to containers for RT serverless edge computing.

***Index Terms*—WebAssembly, Container, Real-time, Safety-critical, Cloud computing, edge computing, Serverless, FaaS**

I. Introduction

As seen in recent EU projects such as SECREDAS [1], a new trend of offloading real-time (RT) systems applications is emerging. Edge computing is becoming popular as it enables data processing on edge nodes that are located only a few network hops away. This minimizes and bounds the worst-case network latency, which is crucial for RT systems.

Previous works [1], [2], [3], suggested fine-tuned versions of cloud hypervisors (e.g., KVM [4]) for real-time cloud/edge nodes, allowing multiple operating systems/applications to run on the same physical machine in Virtual Machines (VMs). Hypervisors provide strong CPU-level temporal isolation and memory-level spatial isolation, in addition to security. However, VMs come with significantly higher resource usage and performance overhead.

Some works [5], [6], [7], [8] have explored containers (e.g., docker [9]) as a lightweight alternative to VMs, especially for supporting RT serverless/Function-as-a-Service (FaaS) computing. Serverless computing allows for automatic scaling of resources based on demand, saving costs and improving efficiency. FaaS allows developers to write and deploy code as individual functions to be executed in response to specific events, and can be used for offloading aperiodic real-time applications to the edge. Periodic RT applications can also use serverless computing to help reduce resource over-provisioning between periodic invocations.

Although containers have much lower runtime overheads (near-native application) compared to VMs, they can still suffer from large (100s of milliseconds) cold startup times (i.e., on first invocation) for initialization and resource provisioning [10]. A way to tackle cold startup problems for RT applications is by over-provisioning CPU and memory resources to keep container (or VM) instances warm between invocations [8], [1], [11]. However, this is contrary to the resource efficiency goal of serverless computing.

Previous work on non-RT serverless edge computing (e.g., [12], [13]) proposed using WebAssembly (WASM) [14], a binary instruction format, due to negligible cold startup times. WASM allows developers to compile code written in programming languages such as C++, Rust, and Go into a portable format that can be executed on multiple hardware and software platforms. WASM also has potential for use in RT systems as it runs code in a sandboxed environment, providing isolation and security like containers/VMs.

However, the WASM runtime environment can introduce additional overhead compared to native/container-compiled code. In addition, WASM is still a relatively new technology and needs more exploration to understand its suitability for RT systems.

Previous works [15], [16], [13], [14] have generically analyzed the performance of WASM runtime compared to containers for IoT or execution as native Linux application or execution of WASM code in the browser. We, on the other hand, use benchmarks to understand the difference between the cold startup times and runtime overheads for WASM runtimes (Wasmtime [17], WebAssembly Micro Runtime (WAMR) [18]) and docker containers. Our study is not about proving one technology's superiority over another. It's about understanding the suitability of Docker container or WASM for different use cases; and if WASM is better suited for a specific use case, we aim to identify which particular WASM runtime is the best for that use case. We performed an experimental evaluation on a server-grade edge node (Dell R640 [19]) with 2^{nd} Gen Intel Xeon Gold Processor [20] (2.3 GHz, 16 cores) and 8 GB RAM running Ubuntu 24.04.

The remainder of the paper is organized as follows: Sect. II presents related work on the suitability of WebAssembly for edge computing. Sect. III gives a brief overview of We-

bAssembly runtimes. Sect. IV describes our tests to compare the performance of WASM runtimes and docker, and presents the results. Sect. V discusses factors in deciding between WASM runtimes and containers. Sec. VI concludes the paper.

II. Related Work

Grosch et al. [21] position WebAssembly (WASM) as an essential enabling technology for designing distributed applications across the Edge-Cloud continuum. They proposed a theoretical edge-cloud framework based on WASM to support real-time applications and meet their safety, timeliness, and reliability requirements. Zaeske et al. [22] integrate a WASM interpreter in a safety-critical ARINC 653 Hypervisor and demonstrate the approach's feasibility by assessing the resultant binary size and performance.

Previous work, such as [23], [15], [13], [12], have proposed using WASM for serverless (edge) computing, esp. to support Function-as-a-Service (FaaS). Gackstatter et al. [13] argue that the high cold-start latencies of containers render them useless to support unpredictable and bursty workloads and may cancel the latency benefits that come with edge computing when serverless functions are deployed. Hall and Ramachandran [12] demonstrated how a WASM-based serverless platform may provide similar isolation and performance guarantees of container-based platforms while reducing average application cold start times and the resources needed to host them. Pham et al. [15], [24] evaluated WASM as an alternative to Docker containers for IoT applications. [15] demonstrated that WASM has a modest performance cost over Docker containers and provides security to applications. Kjorveziroski et al. [23] compared the cold start delays and total execution times of three WASM runtimes: WasmEdge, Wasmer, and Wasmtime to the performance of the containerd container runtime. They showed that WASM runtimes have better results in most tests, and Wasmtime is the fastest runtime among those evaluated. Jangda et al. [16] conduct a large-scale evaluation of the performance of WASM vs. native using the SPEC CPU suite of benchmarks. They showed that applications compiled to WebAssembly run significantly slower. Bosamiya et al. [25] implemented two techniques to produce provable safe WASM code. Their evaluation of these two techniques indicated that WASM can be leveraged to produce provably safe multilingual sandboxing with performance comparable to standard, unsafe approaches.

Contrarily, we use PolyBench/C [26] benchmark to understand better the difference between the cold start times and runtime overheads for WASM using three different runtimes and their different modes. We also compared cold start times and runtime overheads with docker containers. We focus on understanding workloads for which a particular WASM runtime excels or workloads where containers outperform WASM.

III. WebAssembly Runtimes

WebAssembly binaries are run by software called WebAssembly runtimes. There are many existing non-web environment WASM runtimes for executing WASM binaries. In this paper, we present the results of testing two WASM runtimes: Wasmtime, and WebAssembly Micro Runtime (WAMR). We also tested Wasmer, but, since it performed worse than the other two runtimes, we don't show its results to save space.

One difference between WASM runtimes and container runtimes (such as containerd), is that WASM runtimes are responsible for translating WASM binaries so that they can run on the target platform, whereas container images already have natively-compiled binaries.

WASM runtimes can also have different running modes: interpreter, Ahead-of-Time compilation (AOT), and Just-in-Time compilation (JIT).

When running in interpreter mode, the runtime executes the equivalent native instruction(s) for each WASM instruction. With AOT, the WASM binary is compiled into its native equivalent in advance, while with JIT, the compilation is done at runtime.

AOT offers the best performance in terms of startup and runtime execution, but this requires compiling the WASM binary for the intended platform, which trades off some flexibility of WASM. JIT retains this at the cost of some performance overhead, but not as much as interpreter mode, which is why we used JIT for our tests.

Runtimes can also have different compiler backends that affect their timing performance. Wasmtime uses Cranelift as its backend compiler. It aims for fast compilation speeds while generating code that performs almost as well as LLVM or gcc.

WAMR has two JIT tiers: Fast JIT and LLVM JIT. Fast JIT has a faster startup time compared to LLVM JIT, but at the cost of execution time performance. WAMR also supports runtime dynamic tier-up from Fast JIT to LLVM JIT, referred to as multi-tier JIT.

IV. Experimentation

In this section, we present our testing of different WebAssembly runtimes and Docker. The goal of these tests are to determine where WebAssembly is better than containers in terms of total execution time.

All tests were performed on a Dell R640 server with an Intel Xeon Gold 5218 processor and 8 GB of RAM running Ubuntu 24.04. The processor has 16 cores, 22 MB of cache, and has a max turbo frequency of 3.9 GHz but was limited to its base frequency of 2.3 GHz for our tests.

We used the PolyBench/C [26] benchmarks for our tests. It is a collection of benchmarks used in many papers evaluating WebAssembly performance.

We used the Alpine docker image as a base image for our test container. This image is based on Alpine Linux, a small and simple Linux distribution based on musl libc and busybox.

WebAssembly binaries were generated using WASI SDK (wasi-sdk-21), a WebAssembly C/C++ toolchain. Both WASM and native binaries were compiled with clang using the same optimization options (e.g., -O3).

A. WebAssembly runtimes and container comparison

We ran each PolyBench/C benchmark 25 times with the docker container and each runtime's available compiler.

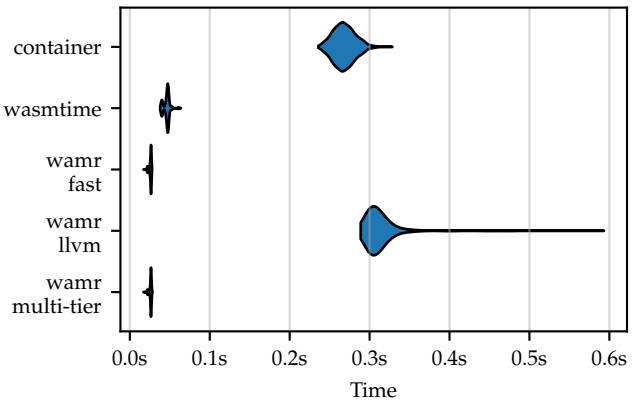


Fig. 1: Startup times.

When offloading RT tasks, the startup time must be included in the WCET, since it delays task completion. Figure 1 shows the startup time results. The startup times were relatively similar for each category regardless of the benchmark. WAMR's fast JIT and multi-tier JIT had the fastest startup times at around 26ms on average, followed by Wasmtime at roughly 46ms. WAMR's LLVM JIT's startup time averaged at around 315ms, which was slower than the container, whose average was around 268ms. It is, however, important to note that the startup time of a container can vary a lot depending on the configuration of the image [27]. Note also that this overhead is only present in the first invocation of a task, since WASM runtimes cache the compiled version of the binary, and containers don't need go through the same setup if they are kept live. Periodic tasks, in particular, only suffer this penalty at the start.

Figure 2 shows the total execution times (i.e., from starting the runtime/container until the end of `main()`) of the runtimes and the container for each benchmark. For benchmarks that last less than a second, WAMR's LLVM JIT and the container performed significantly worse compared to the others. This is largely due to their startup times, which were longer than the total execution time of the others. In most of the longer benchmarks, they outperform the other three.

B. Prolonged benchmark test

The previous test showed that the container and WAMR's LLVM JIT perform better in benchmarks that lasted longer, but this may be due to the workload of the benchmark itself, and not just the duration of the benchmark.

For this reason, we conducted a prolonged benchmark test where we ran the `main()` function of each benchmark multiple times continuously, and noted the time for each run. This simulates a periodic task over multiple jobs, or an aperiodic task with a long duration workload. Doing so allows us to see the trend of how the container and the runtimes perform over a longer time period. The first timestamp in this test is equivalent to the total duration that was measured in the previous test, while subsequent timestamps no longer have a startup period since the WASM runtime/container is not restarted.

To save space, we only show a few selected graphs in Figure 3 which show trends that are representative of what can be observed in other benchmarks. In 8 of the benchmarks, the container had the fastest times after a prolonged run, while in 20 of the benchmarks, either WAMR's LLVM JIT or multi-tier JIT had the fastest times.

V. Discussion

As shown in the previous section, WASM with fast JIT compilers have a clear advantage for short workloads because of its fast startup. This is not a big advantage for longer workloads, where having a more optimized binary is more important, as shown in our prolonged benchmark test. Thus, when deciding on what to use for offloading a task to an edge server, the expected duration of a task should be considered. WAMR's multi-tier approach is of particular interest since it attempts to get the best of both worlds by using a fast compiler at the start and switches to a slower compiler, which is better at optimizing, later on.

Contrary to our expectations, the container did not have the fastest times for the majority of the benchmarks in the prolonged test. This needs to be investigated further since it implies that WASM can perform better than natively-compiled code.

We also want to note that while synthetic benchmarks can be indicative of performance, they don't completely represent real-world workloads. Further tests that run workloads closer to real-world applications are needed.

A. Qualitative Analysis

a) Practical implications: There are other practical aspects to consider when deciding between using WebAssembly and containers.

Even a minimal container image like the Alpine docker image is 5 MB in size, while a WebAssembly binary could be as small as several kilobytes. This has implications on the bandwidth and storage usage of the intended application.

WebAssembly is also platform-independent. Having a single binary for different server platforms that an application may run on is a significant advantage. Edge servers may have different architectures, which necessitates different native binaries that will have their own overheads in terms of maintenance and certification.

b) Limitations of WASM runtimes: WebAssembly in non-web environments is still relatively early in development and does not yet have support for many features.

An example of this is setjmp/longjmp support, which is used in exception handling. This feature has not yet been standardized in WebAssembly, which makes it difficult or perhaps even impossible to use WebAssembly in the many applications that use it.

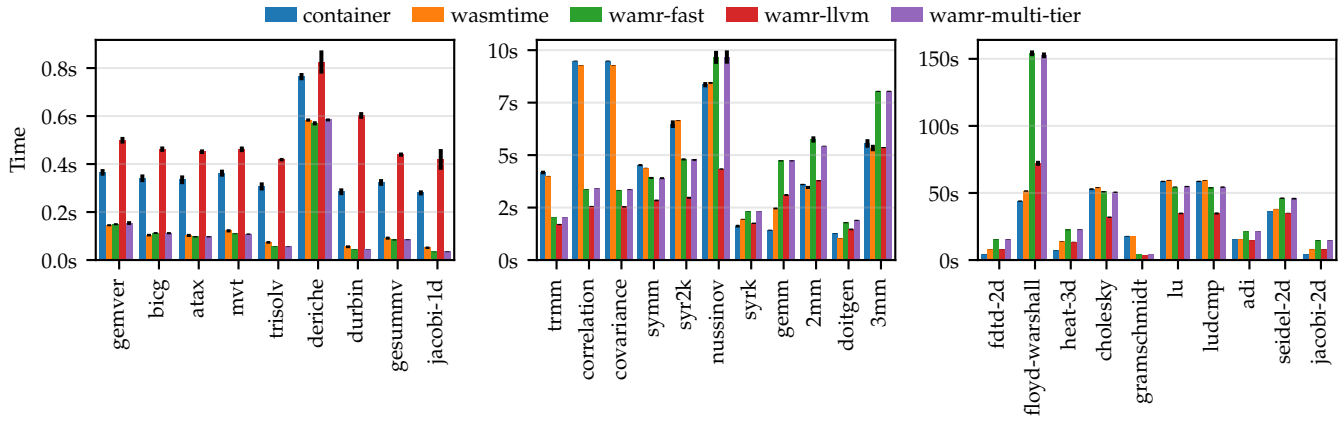
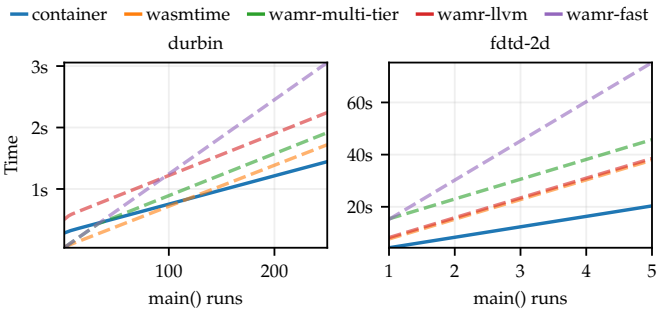
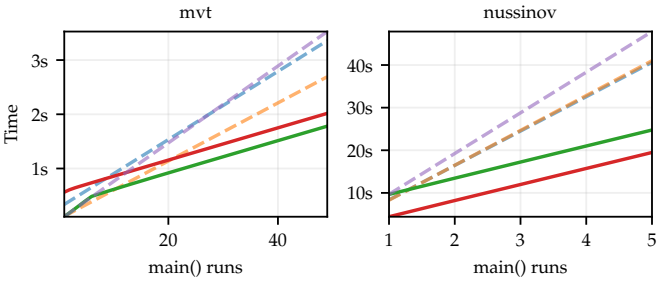


Fig. 2: Total benchmark times.



(a) Container is faster.



(b) WAMR LLVM JIT or multi-tier JIT is faster.

Fig. 3: Prolonged benchmark test results.

Another example is thread support. While the wasi-threads proposal has already been implemented in some WebAssembly runtimes, others have yet to add support for it. This limits the use of some WebAssembly runtimes for multithreaded applications.

These issues, including several others, were encountered when we attempted to compare WebAssembly and containers with a practical application, specifically the You Only Look Once (YOLO) framework used for real-time object detection [28]. We needed to work around these issues to get it to compile to WebAssembly. Due to these workarounds, we were not sure if it was still a valid test. For a sample object-detection test case, the native binary took about 300 milliseconds on average to make a prediction. In comparison, Wasmtime took about 2 seconds on average. WAMR took about 3.5 seconds on average, but it was only spawning 4 processing threads, as opposed to creating a thread for each core similar to Wasmtime and the native binary, even when specifying 12 as the max number of threads. Lastly, Wasmer crashes since it would reach the maximum number of open file descriptors because it would not close files that the application should only temporarily have opened.

As we noted earlier, these issues are likely due to WebAssembly still being relatively young, and most would likely be addressed in the future.

VI. Conclusion

In conclusion, our assessment of WebAssembly runtimes and containers for real-time serverless computing has highlighted the potential of WebAssembly for short periodic and aperiodic tasks at the edge. While shorter startup times make WebAssembly runtimes advantageous for specific applications, the significance of startup times diminishes for longer execution time tasks, where containers with optimized code compilers may offer better performance.

WAMR's multi-tier approach shows promise in addressing performance drop-offs for longer WCET tasks, presenting an interesting area for further research. A possible avenue for real-time serverless edge computing research is to see if it can combine WebAssembly's quick startup advantage with the better long-term performance of native binaries in containers.

When utilizing WebAssembly for real-time serverless computing, it's crucial to consider factors such as platform independence. This is particularly important as the technology continues to evolve and additional essential features are yet to be added.

Future work will focus on exploring why WAMR outperformed the container in prolonged benchmark tests. This investigation is crucial to understand whether this trend holds for different types of tasks. We also plan to extend our testing to real-world application workloads like the ECRTS ARM industrial challenge.

REFERENCES

[1] G. Gala, G. Fohler, P. Tummeltshammer, S. Resch, and R. Hametner, "RT-cloud: Virtualization technologies and cloud computing for railway use-case," in *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*, 2021.

[2] L. Abeni and D. Faggioli, "An experimental analysis of the xen and kvm latencies," in *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2019, pp. 18–26.

[3] ——, "Using xen and kvm as real-time hypervisors," *Journal of Systems Architecture*, vol. 106, p. 101709, 2020.

[4] "Main page," 2023. [Online]. Available: https://www.linux-kvm.org/

[5] V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-time containers: A survey," in *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[6] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *ACM SIGBED Review*, vol. 16, no. 3, pp. 33–38, 2019.

[7] T. Cucinotta, L. Abeni, M. Marinoni, A. Balsini, and C. Vitucci, "Reducing temporal interference in private clouds through real-time containers," in *2019 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 2019, pp. 124–131.

[8] G. Monaco, G. Gala, and G. Fohler, "Shared resource orchestration extensions for kubernetes to support real-time cloud containers," in *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2023, pp. 97–106.

[9] I. Docker, "Docker," *línea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker*, 2020.

[10] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 181–188.

[11] G. Fohler, G. Gala, D. Gracia Pérez, and C. Pagetti, "Evaluation of DREAMS resource management solutions on a mixed-critical demonstrator," in *ERTS 2018*, ser. 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France, Jan. 2018. [Online]. Available: https://hal.science/hal-01700860

[12] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, ser. IoTDI '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 225–236. [Online]. Available: https://doi.org/10.1145/3302505.3310084

[13] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 140–149.

[14] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[15] S. Pham, K. Oliveira, and C.-H. Lung, "Webassembly modules as alternative to docker containers in iot application development," in *2023 IEEE 3rd International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)*, 2023, pp. 519–524.

[16] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of WebAssembly vs. native code," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 107–120. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/jangda

[17] B. Alliance, "Wasmtime: A fast and secure runtime for WebAssembly," last accessed:05/24. [Online]. Available: https://wasmtime.dev/

[18] ——, "WebAssembly micro runtime," last accessed:05/24. [Online]. Available: https://bytecodealliance.github.io/wamr.dev/

[19] May 2024. [Online]. Available: https://www.dell.com/us-en/work/shop/povw/poweredge-r640

[20] May 2024. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/192444/intel-xeon-gold-5218-processor-22m-cache-2-30-ghz.html

[21] F.-J. Grosch, D. Dasari, N. Pereira, and A. Rowe, "Building reliable distributed edge-cloud applications with webassembly," in *Workshop on real-time cloud systems (RT-Cloud)*, 2022.

[22] W. Zaeske, S. Friedrich, T. Schubert, and U. Durak, "Webassembly in avionics : Decoupling software from hardware," in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, 2023, pp. 1–10.

[23] V. Kjorveziroski and S. Filiposka, "Webassembly as an enabler for next generation serverless computing," *J. Grid Comput.*, vol. 21, no. 3, jun 2023. [Online]. Available: https://doi.org/10.1007/s10723-023-09669-8

[24] J. Napieralla, "Considering webassembly containers for edge computing on hardware-constrained iot devices," 2020.

[25] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-Safe multilingual software sandboxing using WebAssembly," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1975–1992. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya

[26] L.-N. Pouchet and T. Yuki, "Polybench/C 4.2.1," *URL: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1*, 2024.

[27] M. Straesser, A. Bauer, R. Leppich, N. Herbst, K. Chard, I. Foster, and S. Kounev, "An empirical study of container image configurations and their impact on start times," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2023, pp. 94–105.

[28] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," 2022. [Online]. Available: https://arxiv.org/abs/2207.02696