Adapting Kubernetes controllers to the edge: on-demand control planes using Wasm and WASI

Merlijn Sebrechts, Tim Ramlot, Sander Borny, Tom Goethals, Bruno Volckaert, Filip De Turck
IDLab, Department of Information Technology (intec), Ghent University - imec, Belgium

***Abstract*—Kubernetes' high resource requirements hamper its adoption in constrained environments such as the edge and fog. Its extensible control plane is a significant contributor to this, consisting of long-lived processes called "controllers" that constantly listen for state changes and use resources even when they are not needed. This paper presents a WebAssembly-based framework for running lightweight controllers on-demand, only when they are needed. This framework extends the WebAssembly System Interface (WASI), in order to run Kubernetes controllers as lightweight Wasm modules. The framework runs these Wasm controllers in a modified version of Wasmtime, the reference WebAssembly (Wasm) runtime, that swaps idle controllers to disk and activates them when needed. A thorough evaluation shows this framework achieves a 64% memory reduction compared to traditional container-based controller frameworks.**

***Index Terms*—Kubernetes, Webassembly, WASI, controllers, operators, edge computing**

I. Introduction

The adoption of new technologies relies heavily on potential resulting cost savings. Financial and environmental cost reductions have been achieved by increasing computational density in cloud computing. Lower latency costs have stemmed from advancements in telecommunications, such as 5G, and the dispersion of cloud to fog, edge and IoT. These two innovation flows have independently led to widely accepted solutions. Combining these two ideas, however, poses a great opportunity. Adopting cloud computing techniques in the edge has the potential to greatly improve reliability, release software faster and expedite operations. Cloud computing utilizes cloud orchestration for optimal resource allocation and server cluster management such as Kubernetes [1], a Google-led open source project inspired by their internal Borg orchestrator [2, 3]. Its extensible architecture parts a cluster in a control plane and a worker plane. Edge environments, which are typically large complex clusters with limited resources, can benefit significantly from having an orchestrator to manage complexity and size.

The primary orchestration targets of Kubernetes, however, are high-resource cloud clusters. Running Kubernetes on low-resource clusters suffers from relatively high control plane overhead costs, which hinders adaptation in the edge market segment. In complex cloud-native application deployments, operators [4] are used to automate actions on the Kubernetes cluster state, that would otherwise be performed by a human operator. These operators are one of the main cost drivers of the Kubernetes control plane. To react to changes in the Kubernetes cluster state, the operators have to run as long-living processes. Even if the operator's control loop is idle, the container and process still use cluster resources. For complex applications that use many operators, these overhead costs quickly accumulate and account for a significant portion of the resource utilization. This is especially problematic for low-resource deployments.

Since global edge application configuration and deployment is a complex task, it is often abstracted by the service provider and included in a Function as a Service (FaaS) offering. FaaS applications are better suited for low-resource edge environments thanks to their fast on-demand scaling properties. Specifically, some edge FaaS platforms use WebAssembly (WASM) [5], a browser technology designed as a portable binary code format that can be assembled from a range of programming languages and that is well suited to resource-constrained environments. On edge FaaS platforms, like Cloudflare Workers [6] and Fastly Compute@Edge [7, 8], WebAssembly is used to securely isolate workloads with reduced overhead and scale-to-zero capabilities.

This research aims to make Kubernetes more suitable to resource-constrained edge environments by running its control plane in a FaaS platform based on WebAssembly. Specifically, the research aims to answer the following questions:

RQ 1: How can Kubernetes use WebAssembly to run parts of its control plane?

RQ 2: How does running the Kubernetes control plane in WebAssembly impact its overhead?

RQ 3: What situations affect the overhead difference between WebAssembly and regular operators?

RQ 4: What is the effect on resource usage of running this control plane in an on-demand manner on a FaaS platform?

Section II investigates the state of the art concerning adapting Kubernetes to the edge. Section III explains the architecture of our WebAssembly-based operator solution and Section IV discusses how this architecture is implemented. In Section V, our benchmark results of the WebAssembly runtime are discussed, as well as the methodology to achieve these results.

II. Related Work

WebAssembly is used by a number of large edge computing platforms such as Cloudflare Workers [6] and Fastly Compute@Edge [7] to securely isolate workloads with reduced overhead and scale-to-zero capabilities. Although Web-

Assembly runtimes generally incur a performance hit compared to native code, this is not inherent to WebAssembly itself, as some runtimes such as Wasmachine show performance *improvements* of up to 21% compared to native code [9]. WebAssembly is especially useful in a Serverless or FaaS setting because of its quick startup times. The Sledge serverless platform, for example, uses WebAssembly to achieve up to 4 times higher throughput and 4 times lower latencies compared to the state of the art [10].

As more and more computation moves towards the edge, so do supporting components such as Kubernetes [11]. Efforts to reduce the footprint of Kubernetes in the edge have shown some success [12]. Although it is possible to reduce the footprint of the existing Kubernetes codebase, the best performance improvements require a complete re-implementation of core Kubernetes components [13]. The overhead of Kubernetes is larger than its core implementation, however, as many microservice deployments and frameworks use custom controllers for management, which add significant memory strain.

Recent additions to the Kubernetes ecosystem, like Krustlet [14] and Wasmedge [15], add support for WebAssembly workloads as an alternative to OCI containers. Efforts to use WebAssembly for extending control-plane components is also underway, such as in the Kubewarden project, which allows writing custom Kubernetes policies in any language and compile them to WebAssembly. [16] Finally, some strides have been made towards using WebAssembly for writing *any* control logic, by creating Wasm Operators [17]. This effort, however, still requires controllers to continuously run, regardless of whether any changes need to be processed. Thus, a solution is needed which combines Kubernetes controllers with WebAssembly and Serverless in order to have a truly event-based Kubernetes control plane which only activates when needed and gets unloaded after execution.

III. Solution architecture: WASM operator

Because of Kubernetes' can-always-fail design, an operator application is not supposed to hold any internal state across reconciliation iterations except for caches. The operator generally uses the Kubernetes API to store state. This theoretically allows running each iteration of the reconciliation loop without storing state in between. This property also holds for FaaS systems, where there is no guarantee for state preservation in between function calls. FaaS solutions for constrained edge environments often utilize Software-based Fault Isolation (SFI) instead of process isolation. WebAssembly lets you create SFI applications based on code written in existing high-level languages.

The *WASM operator architecture* presented in this paper and visualized in Figure 1 attempts to realize all the beneficial aspects that solutions in prior work have achieved [18, 19, 20]. The main parts of the architecture are the parent operator and the child operators. The child operators run as WASM instances in the WASM runtime embedded in the parent operator. We use an existing WASM runtime implementation as embedded runtime. The beneficial aspects are listed below.

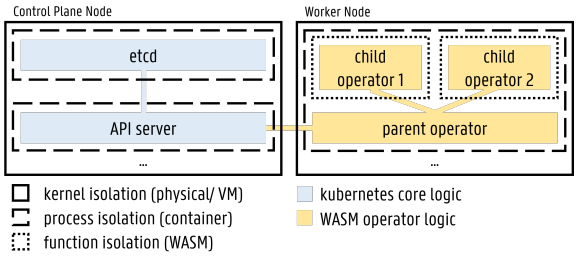


Fig. 1. Design of our WASM operator architecture.

- **Isolation overhead:** All child operators run in the same process as the parent operator. This process runs inside a single container in a single Kubernetes pod. Isolation is provided by the WASM engine, eliminating the overhead due to container isolation.
- **Modularity:** The WASM runtime makes it possible to add or remove child operators without interfering with the other active child operators.
- **Simple child operator:** In our architecture, the parent operator extends the WASM runtime with host functions that can be used by the child operators to communicate with the Kubernetes API. Low-level operator logic is moved to the parent operator. This reduces the complexity and overhead of the client operators.
- **Scale-to-zero:** To limit the overhead of inactive operators, our architecture allows to dynamically unload inactive operators.

In order to efficiently make Kubernetes API requests, we want child operators to perform them asynchronously. Existing WASM runtimes offer no support for asynchronous calls or offer a solution that is incompatible with idle module unloading. Therefore, we created a new solution that adds support for asynchronous operations to the WASM runtime that is embedded in our parent operator. By extending the WASM runtime, we allow the child operators to wait for host functions asynchronously.

A. Parent operator asynchronous runtime

Figure 2 shows how the parent operator manages the asynchronous operations of a child operator and unloads an inactive child operator after a long period of inactivity. The main components of the parent operator are the WASM engine, the host functions exposed to the WASM instance and the work queue. For the WASM engine component and some of the host functions, existing solutions can be used.

The solution works as follows: Each WASM module has an entry point that executes the main function in the child operator, shown in Figure 2 as ①. Environment interactions happen through the calling of WASM host functions ②. Some of these actions are asynchronous and do not directly yield a result. These asynchronous actions are started and added to the work queue ③, directly returning control to the WASM

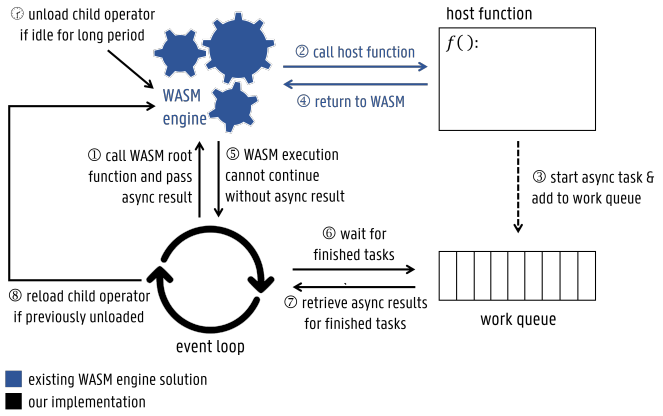


Fig. 2. The design of the parent operator incorporates a WASM runtime event loop which repeatedly performs actions 1-8; making it possible to asynchronously call host functions from within a WASM instance.

module ④. After executing all synchronous logic, the WASM execution stops and the control is returned to the event loop ⑤. This loop checks if any of the actions in the work queue finished ⑥ and passes the results of that finished action ⑦ back to the WASM engine ①, reloading the child operator in case it had been previously unloaded ⑧. When returning to WASM, a new set of synchronous actions are performed by the engine. Long-running operators repeat this process indefinitely. These operators are always waiting for new asynchronous inputs, like events in a watch stream.

The runtime might detect that a certain child operator has not been receiving any asynchronous results over a long period (marked as 🕒). This is indicative for an operator that reached a steady state in its reconciliation process. Most likely, it will only restart its logic after external applications changed the state of the Kubernetes resources that it manages. This could mean that the operator remains idle for multiple hours. In such cases, it can be more resource-efficient to unload and swap the WASM instance to disk.

B. Child operator asynchronous client

All client operators run as single-threaded asynchronous WASM instances. The child operator is started by the host which calls the `start` function that is exposed by the WASM module. This initial function starts the operator reconciliation loop, which makes asynchronous requests. These asynchronous futures [21] are `awaited` by the child operator, but some of these futures `await` asynchronous host function results from the parent runtime. If the child operator cannot continue without new results from the host environment, it stops the execution and returns to the host. If none of the pending asynchronous requests have finished already, the host waits for one of them to finish, as described in Section III-A. Once a request finishes, the host returns the result to the child operator, such that the child operator can finish the linked asynchronous request. This restarts the whole process.

IV. Implementation

We implemented this architecture as an open source runtime, released under the Apache 2.0 license on GitHub [22]. Operators running in this runtime need to use a modified version of kube-rs, which contains the necessary bindings to communicate with the runtime [23].

A. Prior work

Our operator implementation builds upon the proof of concept (PoC) made by Francesco Guardiani and Markus Thömmes [17]. This PoC provides a WASM operator solution based on the Wasmer [24] WASM runtime and a hacked version of the kube-rs [25] library. However, at the time of writing, it has been 2 years since this project was updated. Since the API of the Wasmer runtime drastically changed after its v1 release, and the hacks applied to the kube-rs project are not well documented, upgrading the PoC was not straight forward. Furthermore, the Wasmer project lacks the future potential that other open-source initiatives, like Wasmtime, can offer. To update kube-rs more easily in the future, a new project structure was required. Moreover, the original version of the PoC cannot unload inactive operators as its architecture is different from the architecture proposed in Section III. We refactored the PoC and updated it to implement the aforementioned architecture. Finally, we implemented several improvements to further optimize the PoC implementation, such as adding support for caching compiled WASM modules for later reuse.

B. Parent operator: WASM runtime

The parent operator extends the Wasmtime WASM runtime. Wasmtime was chosen over other WASM runtimes, because it is the flagship WASM engine from the Bytecode Alliance, with support from some of the biggest players in the technology industry. Our implementation configures Wasmtime to compile ahead of time (AOT) new WASM modules to machine code to eliminate the compiler memory overhead at runtime. These compiled modules are cached on disk and can be reused when possible. To initiate these compiled modules, Wasmtime only has to map the file to memory and provide the necessary tools to communicate with this initiated module. Because of the use of file-backed memory, for idle operators, these memory locations can be dropped from memory by the kernel when needed. If the memory region needs to be accessed again, a page-fault will be triggered, and the kernel will load the file back into memory. However, the dynamically populated memory of the WASM module will not be unloaded automatically from memory. That is why our implementation adds a custom unloading and disk swapping implementation in the parent operator. This makes unloading and swapping possible, even on systems without swap enabled at operating system level.

C. Parent operator: host functions

WASM host functions are functions exposed by the WebAssembly runtime to the WASM instances. The Web Assembly System Interface (WASI) is a standardized set of

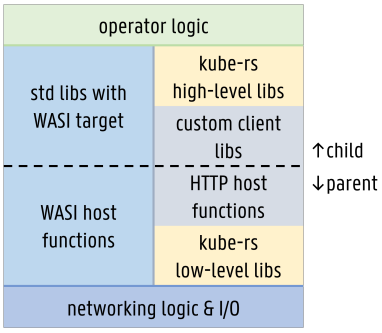


Fig. 3. The operator libraries are split between the parent and child operator and consist out of existing WASI libraries and our own custom libraries based on kube-rs.

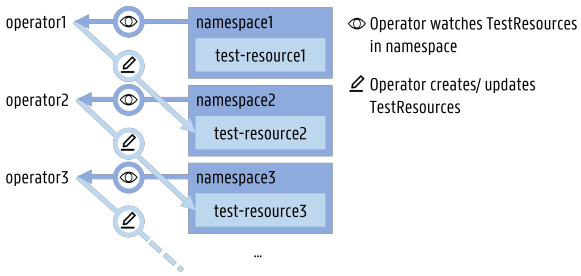


Fig. 4. The test setup for the synthetic-operator workload, each operator is responsible for the propagation of changes from one namespace to another.

these host functions. Our implementation can benefit from the existing Wasmtime library that readily implements these WASI host functions, reducing the implementation and maintenance burden of our solution. A core aspect of Kubernetes operators is communicating with the Kubernetes API server. However, at the time of writing, the WASI spec has not yet standardized sockets as part of the interface [26]. This means that for our implementation, we had to implement custom HTTP host functions to create a working WASM operator setup. As shown in Figure 3, our implementation uses the low-level part of kube-rs for the Kubernetes host function implementation. The high-level kube-rs functionality is implemented in the child operator. The added host functions are asynchronous functions, meaning that they return control to the WASM module immediately, while returning an `async_id` that references a task in the work-queue as described in Section III-A.

D. Child operator: client libraries

All Kubernetes operator domain knowledge is implemented in the custom reconciliation loops, that are defined in the child operators. Our language preference for the child operators is Rust since the Rust standard libraries best support the WASI host function calls. Golang, which is normally used in Kubernetes, has no support for WASI in its default compiler. Additionally, Golang is a garbage collected language, which have been shown to use more memory [27]. Another advantage of choosing Rust as language is that an easier interoperability between the parent and child operator is obtained. The (de)serialization logic mentioned in Section IV-C, can be reused for both the parent and child, since they are both implemented in Rust.

V. Resource utilization

A. Test setup

The test synthetic-operator, as shown in Figure 4, simulates a workload with N different operators, which depend on each other's actions and are idle for most of the time. Each operator watches a namespace for `TestResources` and only reconciles, once a resource is created or updated. It then updates/ creates the resource in its destination namespace. For a full update of all resources, all operators must update their resource one-by-one. This means the full end-to-end latency equals the accumulated individual operator latencies. This synthetic workload simulates a highly dependent and interactive operator setup.

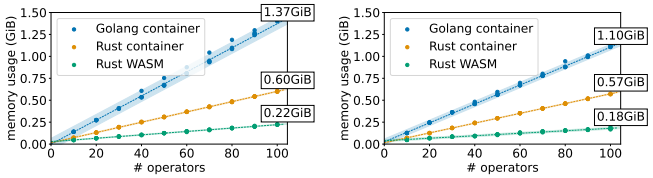
Measuring the memory footprint of a workload execution, requires accounting all the memory usage effects that the process has on the system. This is a non-trivial problem. The memory utilization measure that we use is determined by limiting the memory usage, as determined by cgroup v2 [28], until the application is being slowed down as determined by the Linux PSI metric [29]. For each run, we determine a upper bound memory limit. Each upper bound is defined as the memory limit that is not exceeded for 95% of the selected time range duration. For each configuration, which is defined by an operator type and number of operators, five independent runs were performed, each yielding one upper bound for the active and one for the idle period. Per operator type, we tested the number of operators from 10 to 100, in increments of 10. For the active and idle selection separate, based on the resulting 50 upper bounds for each operator type, we trained a linear regression model. Using this linear model, we determined the 95% prediction interval in which we expect with 95% certainty the upper bound memory usage of a new run with the given configuration, as described by Neter et al. [30].

The end-to-end latency is measured by the synthetic-operator test for the active period of the test. Each set of reconciliations starts from an update of the `TestResource` in namespace 1 until the `TestResource` in namespace N is updated. The time from start to end is measured and each reconciliation set is repeated 500 times per run, resulting in $500N$ reconciliation iterations. As described in Section V-A, for each configuration, which is defined by an operator type and a number of operators, five independent runs are performed.

The raw results of the benchmarks and the analysis and visualization code are available on GitHub [31].

B. Golang container, Rust container and Rust WASM compared

Figure 5 shows the obtained memory upper bounds for container-isolated operators written in Golang and Rust and a WASM-isolated Rust operator. The colored areas represent the



(a) active (b) idle

Fig. 5. The memory 95% upper bounds of the different languages/ isolation techniques are ordered as follows: Rust WASM < Rust container < Golang container; all operators use less memory when idle.

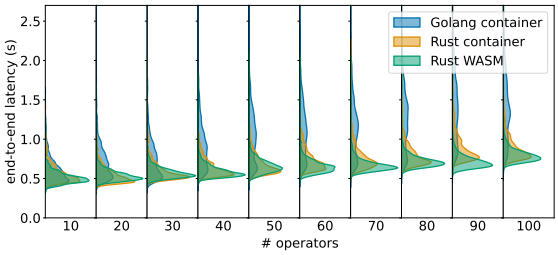


Fig. 6. The end-to-end latency of WASM operators is identical to Rust operators.

95% prediction intervals for the regression models as described in Section V-A.

Figure 5a shows the results for the active period. The Golang-based operator clearly uses the most memory. For 100 active operators, switching from Golang to Rust resulted in a 56.06% upper bound memory reduction. WASM operators even yielded an 83.81% reduction compared to Golang operators. Compared to Golang, the Rust operators use entirely different operator library and framework implementations. Each implementation has its own memory trade-offs, which can lead to large differences in memory usage. Additionally, as discussed in Section V-A, garbage collected languages like Golang, typically are less memory efficient than languages without garbage collector like Rust. The Rust container-based operator and the WASM-based operator share much of their source code. However, the WASM-based operators use less memory than container-based operators. This is due to the reduced complexity of the WASM child operator, as much of its low-level operator logic is moved to the parent operator. Moreover, the different isolation techniques used result in a net reduced isolation overhead, which is further explored in Section V-C.

Figure 5b shows that, as expected, all operator types utilize less memory in case of idle workloads, we observed a 14.21% reduction on average. Compared to 100 idle Golang operators, 100 idle Rust operators utilized 48.04% less memory, which is a smaller reduction than when comparing active operators. However, 100 idle WASM operators still used 83.65% less memory compared to idle Golang operators, similar to the active situation. The smaller reduction in memory usage of container-based Rust operators versus Golang operators is due to Golang experiencing a higher relative reduction in memory consumption when going from active to idle. Based on the typical usage pattern of an operator, which can be idle for a long period of time, it is clear that idle memory usage is important.

In Figure 6, the obtained latency distributions for the different operator types are displayed, which were obtained as described in Section V-A. Based on Jangda et al. [32], WASM performance can be 2.5x slower worst-case compared to native execution. The WASM version of the synthetic-operator, however, did not experience any latency penalties. The latency for the Golang operator increased more than the other operators with increased number of operators. However, this is most likely due to the memory pressuring algorithm that adds more latency to Golang because its less memory efficient. There was no measured useful difference in the average latency between the WASM and Rust implementations that was greater than the measured noise. The main bottleneck in the operator's execution is I/O. Therefore, the latencies that occur in CPU-heavy workloads do not affect the synthetic-operator workload much.

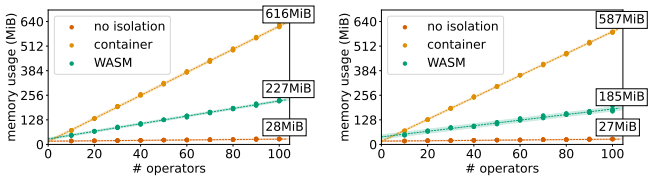
C. Cost of isolation

Figure 7 shows the obtained memory upper bounds for Rust operators using no isolation, using containers and using WASM. The colored areas represent the 95% prediction intervals for the regression models as described in Section V-A.

The solution with no isolation is the most resource efficient. This operator is able to scale to 100 control loops without significant additional memory overhead. Both the WASM-based and container-based setups experience significant per-operator overhead. Additionally, the WASM-based operator has a higher initial constant memory overhead. However, since the container-based solution performs worse per-container, this initial overhead can be compensated. In case of the active situation, the WASM-based solution is more memory efficient than the container-based solution with 95% certainty starting from six operators. For the idle operators, this starts from eight operators.

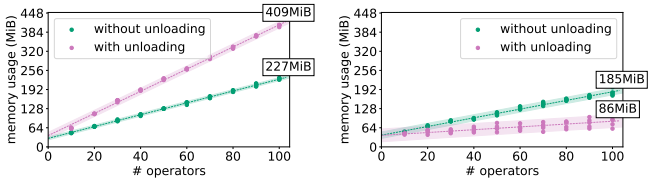
The container-based operators are managed by Kubernetes and each run in a separate Kubernetes pod. Our Kubernetes setup uses containerd [33] to manage the containers. In our tests, the biggest overhead contributor was the per-pod *containerd-shim* process which equates to about 5MiB per pod. The WASM runtime can isolate the modules without introducing such a big overhead. Instead, it introduces a constant initial overhead that does not depend on the number of operators. This memory overhead is due to the WASM runtime, including the low-level operator logic.

Our tests showed that a major memory usage reduction can be achieved by using no isolation. However, having no isolation between operators means that all operators should be fully trusted even for not having errors. Additionally, it results in a lack of modularity: it is not possible to dynamically add or remove controllers. In an operator design based on Kubernetes pods, operators can be added and removed dynamically. Also,



(a) active (b) idle

Fig. 7. The memory 95% upper bounds of non-modular, container-modular and WASM-modular operators show that WASM outperforms container-based isolation, but additional improvements are possible since having no isolation is still much more efficient.



(a) active (b) idle

Fig. 8. The memory 95% upper bounds of the WASM operator with automatic unloading enabled/ disabled; very frequent unloading causes more memory usage, for idle operators it can save memory.

WASM modules can be loaded dynamically by the parent operator, without having to restart the parent operator process. WASM is a good intermediate solution, providing isolation and modularity while still being more memory efficient than the container-based solution.

D. Automatically unloading WASM modules

Figure 8 shows the obtained memory upper bounds for the synthetic-operator running as WASM modules. Two versions of the WASM operator are compared: one does not unload the WASM instances and the other unloads each WASM instance in-between each iteration of the reconciliation loop. The colored areas represent the 95% prediction intervals for the regression models as described in Section V-A.

Figure 8a shows that the effect of constantly unloading and reloading active WASM operator was an 80.49% increase in memory usage for 100 operators. In Figure 9, the effect of actively unloading and reloading operators on the measured end-to-end latencies is displayed, this figure was obtained as described in V-A. Figure 8b shows, running 100 operators, we achieved a 52.66% reduction for idle operators compared to not unloading.

Unloading the modules reduces memory usage in case of idle operators. The parent operator writes the memory of idle WASM instances to disk and reloads it later when a Kubernetes watch event is received, as described in Section III-A. Since most operators often stay idle for a long time, this can greatly optimize resource utilization in memory-constrained environments. However, in case of a worst-case unload and reload pattern, memory usage is higher than in case no unloading and reloading takes place. Frequent unloading also introduces a large end-to-end latency penalty due to the disk overhead of swapping the WASM instance, as shown in Figure 9.

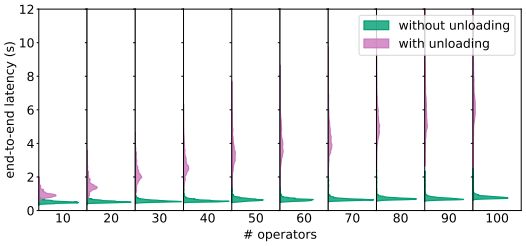
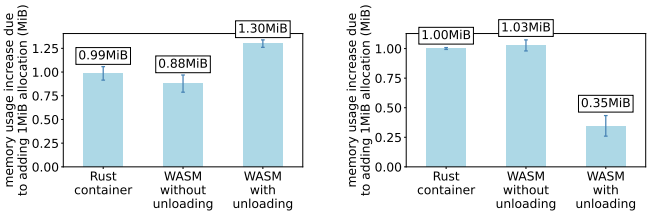


Fig. 9. The end-to-end latency for active WASM operator with unloading enabled/ disabled. Actively unloading and swapping modules introduces significant latency.



(a) The per-operator memory 95% upper bounds increase due to an extra 1MiB of dynamically allocated memory for active operators.

(b) The per-operator memory 95% upper bounds increase due to an extra 1MiB of dynamically allocated memory for idle operators.

Fig. 10. The effects of allocating 1MiB of heap memory in each operator on the 95% upper bounds.

To properly benefit from automatic WASM module unloading in a mixed active-idle situation, a predictive scheduler is a necessity, this is considered as future work in this paper. Such a scheduler could help unloading only when it is beneficial to unload a WASM module instead of unloading it in-between each control loop iteration. The optimization opportunity also greatly depends on the heap memory allocated by the operators, necessary for Kubernetes API state caches. This relationship is further discussed in Section V-E.

E. Dynamically allocated memory

Figure 10 shows the average memory upper bound increase per operator due to a 1MiB increase in dynamically allocated memory. The metric is obtained based on the slope of the linear regression models trained on 20 upper bound memory usage samples obtained for experiments with allocation sizes of 0MiB to 3MiB, with 5 runs per experiments. Also indicated are the 95% confidence intervals for these slopes.

Figure 10a shows that dynamically allocating 1MiB additional heap memory in each operator resulted in in a memory upper bound increase of roughly 100MiB for 100 active operators with unloading disabled, and in an increase of 130MiB for active WASM operators with unloading enabled. The 30MiB extra overhead originates from the additional memory required to reload the WASM module.

Figure 10b shows that the memory consumption for idle operators only increased with 0.35MiB when using our un-

loading and swapping solution. This is significantly lower than the memory increases for operators without unloading and swapping.

As discussed in Section V-D, adding swapping also adds end-to-end latency. For our experiments, it took about 26ms to swap 1MiB of data to disk per operator, which can be fully attributed to the disk read and write overhead of the hard disk drive in the test server. No latency increase was experienced when using the containerized solution or the WASM solution without unloading.

Operators that watch a large amount of Kubernetes cluster resources will typically keep many of these resources in a cache that they update once the Kubernetes API notifies that a resource change took place. This means that these operators have large amounts of dynamically allocated memory, which directly translates to a memory upper bound increase, as discussed in this section. To reduce this memory usage, it is possible to use our unloading implementation in combination with a tuned scheduler. However, such a solution will result in larger latency overhead due to disk writes. Another solution is to move all operator caches to the parent operator and to deduplicate the resources in these caches.

VI. Conclusion

Complex Kubernetes operator workloads are often too heavy for constrained environments. In this article, a novel WebAssembly-based Kubernetes operator solution is proposed. This solution demonstrates that WebAssembly, a technology used by edge FaaS solutions, can also be used to reduce the overhead associated with Kubernetes cluster management. It therefore extends the Wasmtime runtime, adding support for asynchronous Kubernetes API interaction and unloading of idle operators. Our test results show a reduction in memory footprint of 100 active synthetic operators from 1405MiB to 227MiB and of 100 idle operators from 1131MiB to 86MiB by using WASM operators instead of traditional operators. This reduction is due to reduced child operator complexity and the lower WebAssembly isolation overhead. We also found that CPU overhead, identified as a drawback of WASM in prior work [32], does not affect end-to-end latency for our synthetic-operator workload. Unloading WASM operators reduces memory usage for idle operators, while increasing memory usage and end-to-end latency for idle operators. Therefore, future work is needed to add a predictive scheduler that fully optimizes this feature.

Our WASM architecture and implementation demonstrate that initiatives, such as the metacontroller project [19], can integrate a WASM runtime as an alternative to their current WebHook solution and benefit from reduced complexity and resource usage. Resource-constrained edge environments are able to run more WebAssembly operators than traditional operators, enabling complex workloads. Cloud deployments become more resource efficient by replacing existing operators with WASM-based operators. The shared benefits of our solution across both edge and cloud segments help accelerate research and adoption.

The biggest open challenges for developing new WASM operators are the WASM and WASI specifications that are still under development. In addition, Golang lacks proper support for WASI, making it more difficult to write operators in Golang. However, Rust operators can more easily take advantage of running as WASM modules. We further propose to obtain additional reductions in memory usage by moving caching logic from the child to the parent operators.

VII. Acknowledgements

This article is based on the master's thesis of Tim Ramlot [34], co-author of this paper.

References

[1] E. A. Brewer, "Kubernetes and the Path to Cloud Native," in *Proceedings of the ACM 6th Symposium on Cloud Computing*. ACM, Aug. 2015, p. 167.

[2] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale Cluster Management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 18:1–18:17.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade," *ACM Queue*, vol. 14, no. 1, pp. 70–93, Jan. 2016, publisher: ACM.

[4] Shubham, C. Bühler, T. Bannister, and Q. Teng, "Kubernetes Operator Pattern," Mar. 2022. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[5] A. Rossberg, B. L. Titzer, A. Haas, D. L. Schuff, D. Gohman, L. Wagner, A. Zakai, J. F. Bastien, and M. Holman, "Bringing the Web up to Speed with WebAssembly," *Communications of the ACM*, vol. 61, no. 12, pp. 107–115, Nov. 2018, publisher: ACM.

[6] Cloudflare, "Cloudflare Workers®," Mar. 2022. [Online]. Available: https://workers.cloudflare.com/

[7] Fastly, "Fastly Compute@Edge," Mar. 2022. [Online]. Available: https://www.fastly.com/products/edge-compute/serverless

[8] P. Hickey, "Announcing Lucet: Fastly's native WebAssembly compiler and runtime," Mar. 2019. [Online]. Available: https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime

[9] E. Wen and G. Weber, "Wasmachine: Bring the Edge up to Speed with A WebAssembly OS," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, Oct. 2020, pp. 353–360, iSSN: 2159-6190.

[10] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 265–279. [Online]. Available: https://doi.org/10.1145/3423211.3425680

[11] G. Carvalho, B. Cabral, V. Pereira, and J. Bernardino, "Edge computing: current trends, research challenges and future directions," *Computing*, vol. 103, no. 5, pp. 993–1023, May 2021. [Online]. Available: https://doi.org/10.1007/s00607-020-00896-5

[12] V. Kjorveziroski and S. Filiposka, "Kubernetes distributions for the edge: serverless performance evaluation," *The Journal of Supercomputing*, vol. 78, no. 11, pp. 13 728–13 755, Jul. 2022. [Online]. Available: https://doi.org/10.1007/s11227-022-04430-6

[13] T. Goethals, F. De Turck, and B. Volckaert, "FLEDGE: Kubernetes Compatible Container Orchestration on Low-Resource Edge Devices," in *Internet of Vehicles. Technologies and Services Toward Smart Cities*, ser. Lecture Notes in Computer Science, C.-H. Hsu, S. Kallel, K.-C. Lan, and Z. Zheng, Eds. Cham: Springer International Publishing, 2020, pp. 174–189.

[14] Microsoft Research, "Introducing Krustlet, the WebAssembly Kubelet," Apr. 2020, section: posts. [Online]. Available: https://deislabs.io/posts/introducing-krustlet/

[15] WasmEdge Developers, "WasmEdge," 2022. [Online]. Available: https://wasmedge.org/

[16] Kubewarden Project Authors, "Kubewarden: Kubernetes Dynamic Admission at your fingertips," 2022. [Online]. Available: https://www.kubewarden.io/

[17] F. Guardiani and M. Thömmes, "Kubernetes Controllers - A new hope," Jul. 2020. [Online]. Available: https://slinkydeveloper.com/Kubernetes-controllers-A-New-Hope/index.html

[18] D. Srinivas, J. Liggitt, J. Betz, P. Ohly, and others, "kube-controller-manager," May 2022. [Online]. Available: https://github.com/kubernetes/kube-controller-manager

[19] A. Yeh, G. G\ląb, Mike, J. X. Tee, S. Bartscher, L. Villard, and others, "Metacontroller," Apr. 2022. [Online]. Available: https://github.com/metacontroller/metacontroller

[20] C. Ferris and K. Schlosser, "controller-zero-scaler," May 2019. [Online]. Available: https://github.com/ibm/controller-zero-scaler

[21] T. Cramer, xtutu, stephaneyfx, and I. Dmitrii, "The Future Trait - Asynchronous Programming in Rust," Apr. 2022. [Online]. Available: https://rust-lang.github.io/async-book/02_execution/02_future.html

[22] T. Ramlot, M. Thömmes, and F. Guardiani, "Kubernetes Operators in WebAssembly," Sep. 2022, original-date: 2022-09-19T14:37:44Z. [Online]. Available: https://github.com/IBCNServices/wasm-operator

[23] T. Ramlot, "kube-rs patches for wasm-operator," Sep. 2022, original-date: 2022-09-19T14:40:48Z. [Online]. Available: https://github.com/IBCNServices/kube-rs

[24] S. Akbary, I. Enderlin, M. McCaskey, and others, "Wasmer," Apr. 2022. [Online]. Available: https://github.com/wasmerio/wasmer

[25] E. Albrigtsen, T. K. Röijezon, kazk, M. Bagishov, R. Levick, and others, "kube-rs," Apr. 2022. [Online]. Available: https://github.com/kube-rs/kube-rs

[26] D. Bakker and L. Clark, "The WASI sockets proposal," Mar. 2022. [Online]. Available: https://github.com/WebAssembly/wasi-sockets

[27] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," in *Proceedings of the ACM 20th Conference on Object Oriented Programming Systems and Applications*. ACM, 2005, pp. 313–326.

[28] T. Heo, "cgroupv2 memory," Oct. 2015. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html#memory

[29] J. Weiner, "PSI - Pressure Stall Information," Apr. 2018. [Online]. Available: https://www.kernel.org/doc/html/latest/accounting/psi.html

[30] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman, "Applied Linear Regression Models," in *Applied Linear Regression Models*, ser. Irwin series in statistics. Irwin, 2005, section: Chapter 2.6.

[31] T. Ramlot, "Raw experiment data and analysis code of wasm-operator paper and master's thesis," Sep. 2022, original-date: 2022-09-29T13:44:35Z. [Online]. Available: https://github.com/IBCNServices/wasm-operator-results

[32] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code," in *Proceedings of the USENIX 2019 Annual Technical Conference*. USENIX Association, Jul. 2019, pp. 107–120. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/jangda

[33] M. Crosby, L. Liu, P. Estes, D. McGowan, S. Day, A. Suda, and others, "containerd," May 2022. [Online]. Available: https://github.com/containerd/containerd

[34] T. Ramlot, F. De Turck, and B. Volckaert, "Optimising memory usage of Kubernetes operators using WebAssembly," Master's thesis, Ghent University, 2022, publisher: 2022. [Online]. Available: http://lib.ugent.be/catalog/rug01:003063694