

# Optimizing WebAssembly Garbage Collection in Go: Performance Insights, Tuning Tips, and Batch Execution Strategies

Safia Guellil\*, Marc Sánchez-Artigas\*

\*Computer Science and Maths, Universitat Rovira i Virgili  
Tarragona, Catalonia, Spain  
{safia.guellil, marc.sanchez}@urv.cat

**Abstract**— WebAssembly is gaining popularity these days as a portable intermediate binary format for programming languages. With a well-specified low-level virtual instruction set, low memory footprint and high-performance virtual machines, it has arisen as a lightweight alternative to containers. Even the tiniest container typically sits somewhere around tens of MBs and requires several hundred MBs of memory to run, which is infeasible in resource-constrained edge devices. Though WebAssembly has been battle-tested for memory *unmanaged* languages such as C++, there exists a systematic lack of investigation on *managed* languages such as Go that utilize a garbage collector (GC). Taking TinyGo as a paradigmatic example, we study in this paper the implications of garbage collection for WebAssembly. We show that WebAssembly introduces some inefficiencies compared to native execution and share tips on how to raise its performance. Yet more compelling, we demonstrate that it is possible to dynamically adjust the GC by rewriting the WebAssembly binaries, and in this way, control the impact of garbage collection on the execution time by trading off extra memory. In edge contexts, this ability is essential to fine-tune the execution of a batch of WebAssembly jobs over the same physical device as shown here for the first time.

**Index Terms**—WebAssembly, Go, garbage collection, edge

## I. INTRODUCTION

WebAssembly has recently become a very popular bytecode language that claims near-native speeds, fast startup times, and a low-level memory model [1]. First developed for the web, it has gained momentum in the server space over the last years as an alternative to containers [2], finding its way not only in the public cloud [3], [4], but also in the edge [5]–[8]. With this, we do not intend to mean that containers do not work well at the edge at a certain scale. However, *what if one wished to quickly spin up applications on a Raspberry Pi?*

Although it is possible to optimize Docker containers, even the smallest container would sit somewhere around hundreds of megabytes, and execute in the GBs of memory per instance. What this means is, at the far edge, containers may be too large for the resource constraints, even at its lowest denominator. In this sense, there is a huge interest to find lightweight ways to “pack” as many instances as possible with the same resources.

Moreover, even highly optimized containers can exhibit cold start times measured in seconds. Consequently, containerized applications that need to quickly respond to requests in a few milliseconds have no other way to be predeployed and run idle

before requests touch the device, thereby consuming valuable resources.

Luckily, WebAssembly (abbrv. Wasm) small size and secure sandbox enable fast code execution anywhere in the computing continuum [8]. With a cold start time in the range of hundreds of microseconds [9], WebAssembly effectively overcomes the “cold start problem”. Its tiny size means that it can be scaled to a higher density than containers and run on demand [5], [7].

Though the performance of WebAssembly has been studied well with memory unmanaged languages such as C and C++ [1], [10], there exists a systematic lack of research on managed languages like Java and Go that have a ‘garbage collector’ (GC) to automatically take care of memory management. This lack of investigation is not only worrisome given the popularity of these languages, but also a missed opportunity. For instance, an efficient garbage collection can further reduce the memory footprint of a series of WebAssembly instances to make them fit in a memory-limited edge device.

At the time of this writing, there are two major approaches for garbage collection in WebAssembly:

- *Compilation of the original GC into WebAssembly*, where the native implementation of the collector is compiled to WebAssembly bytecode; and
- The brand-new *WebAssembly Garbage Collection (abbrv. WasmGC) proposal* [11], where some of the data structures in the source language (e.g., Go) are compiled down to the available GC constructs in WebAssembly.

As of today, the support for the WasmGC proposal is limited to only Web browsers, Chrome and Firefox; and it is at an early stage, with lack of support for important features like interior pointers [11], which for instance prevent compiling Go code to WasmGC.

For this reason, we examine in this paper the only practical solution until today for server-side execution: the compilation of the collector code into the WebAssembly language, and its subsequent inclusion on the binary. Apparently, this approach has the main flaw of increasing the size of the WebAssembly binaries, but it has also important benefits. The most laudable is the ability to preserve the garbage collection optimizations of the source language, along with the opportunity to fine-tune garbage collection in the binaries themselves, without external support from the WebAssembly virtual machine.

**Why Go?** For our investigation, we chose Golang (a.k.a. Go), because it is one of the most popular languages for developing microservices, and has a battle-tested support for edge devices via the TinyGo [12] compiler. Concretely, most of this research focuses on the TinyGo compiler, because the cross compilation of Go applications with the official Golang compiler produces large Wasm binaries that perform poorly. Actually, one of the negative aspects of compiling the GC logic to WebAssembly is that during GC runtime, the entire application must be paused, a phenomenon known as “stop-the-world” (STW) [13] in other languages such as Java, due to the non-concurrent execution of the Wasm bytecode in the Wasm virtual machine (VM). This is further aggravated by the fact the Go GC is “non-incremental”, which poses an interesting trade-off between keeping memory consumption at bay or degrading performance. But, “the story does not end here”. WebAssembly provides no means to return free memory back to the operating system (OS) once the heap is extended, and uses a page size of 64KiB, namely, 16x bigger than a regular Linux page size. Altogether, this change of rules makes the study of garbage collection in WebAssembly a new avenue of research. Just to illustrate, it poses questions like:

- *Do Wasm-compiled Go applications suffer severe runtime degradation compared to running the native binary on the target architecture (e.g., x86)?*
- *How does the Wasm-compiled GC perform in comparison to the native one?*
- *What are the practical implications of running Wasm-Go programs on edge devices like a Raspberry Pi?*

We respond to these questions for the first time ever. But most importantly, we demonstrate that our new insights can provide practical advantages to WebAssembly execution. Specifically, we prove that Wasm-Go programs can be significantly sped up by decreasing the number of GC interruptions at the expense of a higher memory footprint. Curiously, the implementation of this idea involves the instrumentation of the Wasm bytecode. Because TinyGo statically embeds the GC logic into the Wasm modules, we dynamically rewrite their bytecodes to tweak the GC and boost their performance. Simply put, our view is that from an edge provider that executes Wasm modules and has no access to the original source code. This model emulates that of AWS Lambda for compiled languages such as C++, Rust and Go, where the code must be compiled natively ahead-of-time (AOT) by the end user and only the final binary is uploaded to the cloud [14]. Sticking to this model, our results report latency improvements larger than 2.2x in a Raspberry Pi 4, showcasing the benefits of our universal, black-box approach.

**Contributions.** In summary, this paper contributes:

- The first deep study of garbage collection for server-side WebAssembly using Golang. Our main goal is not only to comprehend the impact of the Wasm execution model on Go programs, but to understand how it can restrict their performance due to the action of the “Wasmified” GC.
- An extensive evaluation of the overheads introduced by Wasm on garbage collection using a number of standard Go benchmarks.

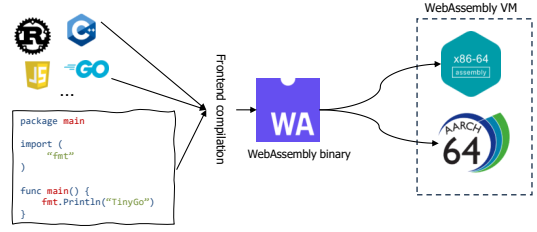


Fig. 1: A holistic view of Wasm compilation and execution.

- A demonstrative example on how learning how to tweak the GC straight in the Wasm bytecode via binary rewriting can be a valuable tool to control the performance of Wasm containers over small edge devices.

**Open source and data.** Our code will be made available under an open source license.

## II. BACKGROUND ON WEBASSEMBLY

**Overview.** Fig. 1 illustrates a holistic view of the typical Wasm compilation and execution pipeline. When a program, written in a high-level language like C/C++, Rust, JavaScript, Golang, etc., is compiled to WebAssembly format, the resulting binary is called “module”. After that, the module can be executed on any of the supported platforms (e.g., x86\_64) with a standalone runtime or virtual machine (VM) [1] such as Wasmtime [15].

The WebAssembly VM is a stack machine without registers. This means that the instructions pop their inputs from and push their output results to the implicit stack. There are 4 primitive data types: 32-bit and 64-bit integers and floats. Complex data types such as C/C++ structs do not exist. Their source-level types are encoded as raw bytes and saved to the linear memory.

Besides traditional control flow instructions, e.g., if-else, WebAssembly adds specific ones. For instance, `br_table` can define multiple target branches to jump to. Further, it must be noted that WebAssembly features Control Flow Integrity (CFI) to prevent exploits that attempt to take control of program flow.

**Memory.** WebAssembly does not provide “managed” memory or garbage collection by design. Instead, the standard proposes a byte-addressable linear memory region and `load` and `store` instructions to access arbitrary addresses within this memory. For this purpose, Wasm uses 32-bit pointers, where the “i32” integer type serves as the pointer type. This limits the available memory of a module to only 4GBs, but also enables ultra-fast bound memory checks at runtime [16], which is basic to grant Software Fault Isolation (SFI) [17] capabilities to modules. For effective dynamic memory allocation, each frontend compiler embeds a memory allocator of its choice. For instance, TinyGo provides a block-based allocator. The compiler is responsible for arranging the memory layout by breaking down the linear memory into a call stack, heap, and the static data region. The TinyGo memory layout is similar to that of Clang [18], where the shared heap is on top [10] so that it can freely grow towards higher memory addresses.

**WebAssembly System Interface (WASI) [19].** In a nutshell, WASI is a standard API that enables WebAssembly modules to

TABLE I: **TinyGo** 0.32 vs. **Go** 1.22 for “helloworld” [21] compiled to WebAssembly on a Raspberry Pi 4.

	Binary size	Compile time	Exec. time	Instruct.
Go	2.1 MB	15.57s	1.72s	667,347
TinyGo	608.2 KB	14.99s	0.21s	84,886

TABLE II: **Code footprint** of TinyGo GC in “helloworld” [21]. Absolute module numbers are within brackets.

Binary size	LOC (.wat)	Instr.	Funct.	Calls
<b>45 KB</b> (608.2 KB)	<b>9,286</b> (86,984)	<b>9,261</b> (84,886)	<b>18</b> (228)	<b>77</b> (1,308)

interact with the underlying OS, e.g., to perform I/O operations outside browsers. To compile Go code with WASI, developers need to install `wasi-libc` [20], a libc for WebAssembly built on top of WASI syscalls. This libc library replaces most of the syscalls to the underlying OS by the equivalent WASI hostcalls imported from the standalone VM (or runtime).

### III. GO COMPILERS COMPARISON FOR WASI

As a first result, we wanted to analyze the impact of the two main Go compilers with WASI targets. To give some numbers, it sufficed to compile the “Hello, World!” application from the Programming Language & Compiler Benchmarks (PLCB) [21] to WebAssembly with both Go compilers and statistically analyze the binaries.

As reported in Table I, the mainline Go compiler produces bloated binaries for WASI targets. The main insight is that both compilers statically embed the GC logic into the Wasm binary, which is overly “complex” in standard Go for WASI execution. We say “overly complex” for WASI because much of the GC logic in mainline Go is aimed at handling concurrency in order to enable the GC to run alongside the application. The problem is that Wasm execution is single-threaded. And consequently, a concurrent GC behaves as a STW collector in practical terms. Loosely speaking, an application running on Wasm behaves as a Go program running with `GOMAXPROCS=1`. This implies that the scheduler has some margin to run goroutines concurrently, but WASI calls, or any other host function calls, may cause all goroutines to block until the call finishes. An “overengineered” runtime is what makes mainline Go to perform badly in WASI, where the execution time is 8x longer than TinyGo.

In contrast, the TinyGo GC has been code-size-optimized, and its STW semantics are more aligned with the nature of the Wasm single-threaded execution model. This is trivial to see in Table II, where the code footprint of the TinyGo GC is given. The implementation of the GC amounts to only 7% of the total binary size. This lightness is the major reason why TinyGo is better positioned for WebAssembly. The paramount question is then: *how does the WebAssembly execution of the GC perform compared to native execution?*

### IV. DISSECTION OF TINYGO GC

#### A. Overview

Compared to mainline Go GC, TinyGo GC is overly simple as it was designed for embedded systems and microcontrollers with limited computation and memory capacities. In a nutshell,

it is a textbook implementation of the mark&sweep algorithm, with all its benefits and disadvantages. In practice, this means that Wasm execution will inherit its well known inefficiencies, and perhaps, introduce new issues due to the specifics of Wasm execution. For instance, TinyGo WASI programs may struggle with excessive heap fragmentation. Because mark&sweep GC does not move objects, even if the total free space in the heap is sufficient [22], [23], the WASI program may fail to allocate a contiguous region for an object. Further, TinyGo GC supports full GC only. That is, garbage collection operates on the whole heap and cannot be paused or stopped during a GC cycle.

However, the big difference with mainline Go is that TinyGo GC is a fully “stop-the-world” (STW) GC [13]. Consequently, GC may spend significant time to run both its mark and sweep phases, during which the program, either native or WASI, will be unable to make further progress. But a STW GC has its own benefits too, specially when WASI targets are at the forefront. Concurrent execution of Wasm code within the same thread is still not possible outside browsers. So, a concurrent GC such as that of mainline Go will enjoy no latency gains but experience high penalties due to extra synchronization. Non-concurrency brings another upside. No mutator will allocate faster than the GC can mark, thereby preventing an unbounded heap growth. For a 32-bit environment such as WebAssembly, this is crucial to escape premature out-of-memory (OOM) errors.

**Observation 1:** TinyGo garbage collector fits well the single-thread WebAssembly execution model, it is code-optimized, and ideal for small edge devices.

#### B. Methodology

To analyze and characterize the performance of the TinyGo GC, we will use Wasmtime 22.0.0 [15] as the WASI runtime. For JITting WebAssembly code to native code, Wasmtime uses Cranelift [24], a fast code generator that outpaces LLVM by an order of magnitude [25]. We will leverage TinyGo 0.31.2 (Go version go1.22.2) to front-compile Go code to WebAssembly. This version of TinyGo uses WASI-SDK 20. In this work, we only evaluate the conservative variant of the GC. The “partially precise” variant was unavailable for WebAssembly at the time of this writing.

**Benchmarks.** We make use of two standardized benchmarks to identify the idiosyncrasies of WebAssembly GC: the PLCB benchmark suite [21] and several memory-intensive programs from the BigCache library (bigcache-bench) [26].

**Setup.** All tests are performed on a **Raspberry Pi 4** model B equipped with a quad-core ARM Cortex-A72 CPU 64-bit SoC @1.8GHz, 4GB of RAM, and with Ubuntu 22.04.4 LTS as the operating system (OS). Also, we use MacBook Air (8-core M1 CPU, 8GB of RAM) to assess the performance of our solver.

#### C. Heap management

To better shoulder the impact of WebAssembly on garbage collection, here we describe how TinyGo manages the heap.

**Heap layout.** For WebAssembly compilation target(s), TinyGo utilizes blocks of 4 **words** or pointer sizes as units of work.

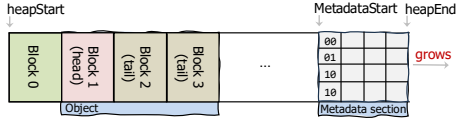


Fig. 2: Heap layout in TinyGo.

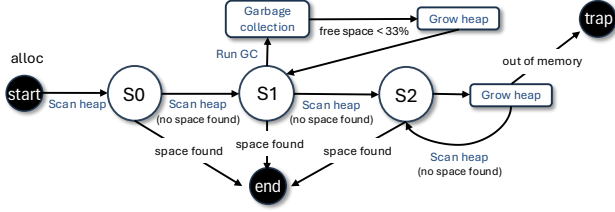


Fig. 3: Heap allocation in TinyGo.

As defined in the MVP, WebAssembly is a 32-bit architecture. So, each block in the heap occupies exactly 16 bytes. Further, each block has associated some metadata that records the state of the block: *free*, *head*, *tail* and *mark*. Initially, all blocks are in *free*. Every allocated object begins with a *head* block, followed by *tail* blocks. Such a head-tail distinction makes it easy to find the start and end of an object. Finally, a block can only be found in the *mark* state within a GC cycle.

This 2-bit metadata per block (00-*free*, 01-*head*, 10-*tail*, 11-*mark*) is stored in a designated area at the end of the heap. Concretely, these metadata is stored in the area delimited by the pointers `metadataStart` to `heapEnd`, while data blocks fill the area from `heapStart` to `metadataStart` (see Fig. 2).

One clear disadvantage of the above scheme is that the size of the metadata region depends on the number of blocks in the heap. Consequently, when the heap grows, the metadata region needs to increase in the same proportion to record the state of each data block. This operation is expensive due to the need to relocate the old metadata via a `memory.copy` instruction.

**Object allocation.** Object allocations into the heap are handled by the `runtime.alloc` function in three steps as illustrated in Fig. 3. For sake of notation, we refer to these steps as S0, S1 and S2. Since the allocation policy is “next-fit” [27], each state could involve an entire traversal of the metadata region of the heap to find the necessary number of contiguous free blocks<sup>1</sup>. In any state, allocation will terminate as soon as the first free segment large enough is found. The object is finally allocated by setting the first block to *head* and the following blocks (if any) to *tail*. Allocation of an object proceeds as follows:

In state S0, there is a first scan of the heap metadata. If the allocation request is unfulfilled, the `alloc` function transitions to state S1. In this state, it starts a new garbage collection cycle to reclaim the memory of inactive objects. After completion of the GC cycle, `alloc` checks if the available space in the heap is below a certain hardcoded threshold  $H$ . If there is insufficient free space, it extends the heap accordingly. Irrespective of this, it transitions to state S2 and rescans the heap metadata to find a segment that is sufficiently long. In state S2, `alloc` repeatedly

<sup>1</sup> Upon an allocation request, the allocation size is rounded up to the nearest block size to ensure block alignment.

grows the heap and rescans the metadata until either fulfilling the allocation or trapping with an OOM panic.

From the above description, it is readily evident that heap metadata scans needs to be time-efficient. Cache locality plays a critical role in this respect. For the WebAssembly target, we observe that one single metadata block can record the state of 64 contiguous data blocks, which clearly favors cache locality. That is, accessing the word that stores the 2-bit metadata of a given block will automatically bring the metadata of the other 63 blocks into the data cache. To verify that metadata scans are quick, we utilized `cachegrind` from `Valgrind` to perform elaborated simulation of Level-1 (L1) and Last-Level (LLC) caches for a number of memory-intensive programs from the Bigcache and PLCB benchmarks compiled to WebAssembly. Overall, cache miss rates were consistently below 0.2%, which signals no inefficiencies in the cache for metadata scans. Only `pidigits`, a CPU-bound program that calculates the digits of Pi, reported an L1 miss rate of 0.55%.

**Allocation issues.** Albeit simple, this allocator has some issues and concerns, some of them innate from WebAssembly. First, the allocation time is proportional to the current heap size. As the heap increases, so does the time spent in finding a segment. This is not the full picture, as the allocation time can be further increased by excessive heap fragmentation. The question here is to determine whether WebAssembly can artificially increase external fragmentation to a level where allocation requests that would be fulfilled in the native binaries cannot be served in the equivalent WebAssembly modules. To check this, we rewrote a part of the `runtime.alloc` function to measure the amount of external fragmentation and its impact on garbage collection. In more detail, the purpose of the added code lines was to identify those allocations that caused the intervention of the GC despite having enough free space. We counted each of these events as an “avoidable” garbage collection cycle. For every avoidable cycle, we then computed the degree of fragmentation (DoF) as 
$$\text{DoF} = 100 \times \left( 1 - \frac{\sqrt{\sum_{\text{seg} \in \mathbb{F}} |\text{seg}|^2}}{\sum_{\text{seg} \in \mathbb{F}} |\text{seg}|} \right),$$
 where  $\mathbb{F}$  denotes the set of free segments, i.e., a contiguous region of free blocks, and  $|\text{seg}|$  denotes the size of segment `seg`. Note that DoF equals to 0% when there is no fragmentation, but takes a value close to 100% when the size of all free segments is just 1 block, i.e., maximal fragmentation. As a global statistic, we also measured the ratio of avoidable cycles to the total number of collection cycles, i.e., **probability of avoidable cycle (PAC)**.

Table III gives the fragmentation results for both the native and WebAssembly execution. Overall, we observe that the vast majority of collection cycles are induced by fragmentation, and are thus avoidable, as shown by the PAC metric. Interestingly, PAC values are higher in native. This is due to the larger size of WebAssembly memory pages (64KB) compared with Linux (4 KB), which facilitates the location of free segments after every new heap expansion. On the contrary, the DoF scores are much higher in WebAssembly, an unequivocal sign of a (much) more fragmented heap. In conclusion, whereas to a certain extent a larger memory page decreases the number of avoidable cycles, the heap has more fragmentation in WebAssembly. Needless to

TABLE III: **Fragmentation.** PAC(%) stands for ‘probability of avoidable garbage collection cycle’ in percentage. Avg(DoF) stands for the average degree of fragmentation.

Binary	WebAssembly		Native	
	PAC(%)	Avg(DoF)	PAC(%)	Avg(DoF)
<i>bigcache</i>	73.91	93.91	86.09	66.41
<i>freecache</i>	85.19	94.60	94.81	91.47
<i>json-serde</i>	62.50	93.86	95.12	92.66
<i>lru-linked-map</i>	86.67	61.17	95.91	67.38
<i>lru-ordered-map</i>	40.74	51.34	30.87	46.97
<i>merkletries</i>	52.00	55.81	6.25	37.71
<i>pidigits</i>	98.71	75.67	99.57	65.11
<i>stdmap</i>	84.62	98.42	99.99	55.28

TABLE IV: **Relative heap growth** beginning with 128 KiB (or with two WebAssembly memory pages) at iteration  $t$ .

$t$	1	2	3	4	5	6	7	8
$R_t$	1.50	2.26	3.40	5.11	7.69	1.56	17.39	26.15

say, this definitely contributes to increase the allocation time.

**Observation2:** WebAssembly execution increases heap fragmentation.

Second, to extend the heap for WASI programs, the TinyGo compiler is forced to emit the `memory.grow` instruction. This instruction expands the linear memory by a specific number of WebAssembly pages, which forces the heap to grow in page-sized increments of 64 KB (WebAssembly page size). At first glance, this may seem irrelevant, but combined with the policy of how much free space left in the heap for the next allocations, it may lead to high memory underutilization. More specifically, the inventors of TinyGo hardcoded the value of the free space threshold to  $\frac{1}{3}$  to ensure at least 33% headroom in the heap at all times. Due to the larger page size, the enforcement of this policy in WebAssembly means doubling the current heap size. Recall that the maximum amount of memory addressable by a WebAssembly module is limited to 4GB or 65,536 pages. One can then expect that around 16 iterations, the module instance will run OOM and be killed by the WebAssembly VM.

To better understand this, let us denote by  $H_t^W$  the heap size of a WASI program after the  $t$ -th heap expansion. Let  $H_t^N$  be the heap size of the same TinyGo application running natively. Then,  $R_t = \frac{H_t^W}{H_t^N} = \frac{2H_{t-1}^W}{H_{t-1}^N + (\frac{1}{3}H_{t-1}^N)}$ , where  $R_t$  denotes the relative heap growth ratio of Wasm to native execution. As shown in Table IV, the heap size in Wasm is 26x larger than native after the 8th heap expansion. This implies that TinyGo programs compiled to Wasm can significantly waste memory.

**Observation 3:** The heap expansion rule in TinyGo is ill-suited for WebAssembly, potentially leading to high memory under-utilization.

#### D. TinyGo GC

As already introduced, TinyGo implements a mark&sweep garbage collector. As such, it consists of two phases: the mark and sweep phases. In the mark phase, all live (or “reachable”) objects from the root set are “marked”. In the sweeping phase, the blocks belonging to unmarked objects are freed. In general,

both phases are textbook implementations with some minimal modifications. To wit, marking a reachable object in TinyGo is nothing but setting the state of its head block to marked, while sweeping the heap simply means setting the state of all blocks from unmarked objects to free. In this sense, TinyGo does not release reserved memory to the OS in the sweep phase. It just remarks the blocks to free to enable their future reuse.

Above all, we see that the use of a “textbook” mark&sweep collector makes the results easier to interpret and to extrapolate to other (STW) collectors. Certainly, the sweep phase is well-optimized for WebAssembly due to its default single-threaded execution semantics. The same terms apply to the mark phase, except that it can be further optimized by determining the right mark-stack size, a tuning knob that is program-dependant and whose default value heavily hurts WebAssembly (see §IV-E).

**Mark phase.** As noted above, the mark phase marks objects in memory that are still reachable in the object graph and in use. In this sense, the mark phase has anything special to tell. The only significant problem of the TinyGo implementation is with the mark-stack, which is **fixed-sized** and as small as 16 blocks for 32-bit architectures. Following best practices, TinyGo GC performs depth-first search (DFS) on the root set to determine all the reachable objects. Specifically, the GC calls the function `startMark(root)` for every root object to start the marking procedure on this root and all of its children. This has the well-known advantage of improved spatial locality, as DFS places the root and its children closer in the cache memory [28], [29]. However, DFS tracing needs a LIFO mark-stack to temporarily store the children blocks yet to be scanned. The problem is that the mark-stack size poses a trade-off for each marking run. A large mark-stack uses more cache lines, more TLB entries, and more time wasted due to cache misses [30]. In contrast, a small mark-stack could cause a mark-stack overflow, which happens when the mark-stack is used up and cannot store more blocks. Unless something is done, the new objects found in the current marking run will left unmarked and mistakenly collected in the sweep phase, leading to **false negatives**.

**Mark-stack overflows.** To deal with overflows, TinyGo adopts a simple strategy: it marks the traversed objects so far, but does not push them onto the mark-stack. In this way, it prevents the occurrence of false negatives. However, this strategy needs of a recovery routine that retraces all the marked objects to ensure no reachable object is accidentally reclaimed. Since the mark-stack is also prone to overflows during recovery, the process is repeated until it can be completed without overflows. We will thenceforth refer to this recovery process as “finish marking”. As above, another key question is to determine whether or not WebAssembly as a compilation target may have some negative impact on the finish marking step, which we answer next.

#### E. Performance tips and results

To capture the garbage collection differences between native and WebAssembly, we conduct a comprehensive analysis and lay out several vital observations related to garbage collection



TABLE V: **Internal GC indicators.** For cross-comparison, the numbers within brackets correspond to native values.

Binary	Avg. no. heap expansions		GC cycles	SO-rate
	Phase S1	Phase S2		
<i>bigcache</i>	9.1 (17.2)	1 (0)	31 (98)	1.9 (0.9)
<i>binary-tree</i>	5 (13)	1 (0)	26 (54)	1.1 (0.2)
<i>freecache</i>	6.5 (14)	3 (5)	42 (68)	1.1 (0.9)
<i>json-serde</i>	1 (2)	6 (12)	8 (21)	1.5 (1.0)
<i>lru-linked-map</i>	0 (1)	1 (0)	1053 (2869)	1.0 (0.1)
<i>lru-ordered-map</i>	0 (1)	1 (0)	1083 (2964)	1 (0.6)
<i>merkletrees</i>	7 (21)	1 (0)	24 (39)	1.8 (0.4)
<i>pidigits</i>	6.7 (2)	4.3 (5)	645 (1328)	1 (0)
<i>stdmap</i>	12 (0)	1 (0)	13 (7022)	2 (0)

time, and its per-phase breakdown: mark, finish, and sweep, as well as the impact of its two main tunable parameters: the heap threshold  $H\%$  and the mark-stack size  $S$ . For this, we run our 9 memory-bound benchmarks and collect numerous metrics. To provide meaningful average results, we run each test 10 times.

**Default configuration.** Here we study the differences between both compilation targets using the default configuration, which is ( $H\% := 33\%$ ,  $S := 32$  blocks) for native, and ( $H\% := 33\%$ ,  $S := 16$  blocks) for WebAssembly. That is, the native mark-stack can temporarily store the double of objects yet to be marked.

Fig. 4 reports the garbage collection time as a percentage of the total execution time, shown on top of the bars, along with the breakdown of phases. The plotted line shows RSS memory consumption. This figure unveils several critical insights. The most significant is the increase in garbage collection overhead. For instance, *stdmap* escalates from 27% overhead in native to 86% in WebAssembly, a 3-fold increase. To a lesser extent, this pattern repeats for the rest of benchmarks. The reasons for this are multi-fold. The main reason is the mark-stack size. A larger mark-stack means less stack overflows, and consequently, less calls to “finish marking”, resulting in improved performance. This is visible by looking at the “SO-rate” column of Table V, or the ratio of stack overflows,  $SO$ , to the total number of GC cycles,  $|cycles|$ , i.e.,  $SO\text{-rate} := \frac{SO}{|cycles|}$ . Despite the higher number of collection cycles in the native execution (Table V), the lower SO-rate makes “finish marking” to be less relevant. This, however, kills WebAssembly performance, as plotted by the larger yellow portions of the bars in Fig. 4.

Another key performance metric is memory usage. Table V reports the average number of times that `memory.grow` was called to expand the heap at stages S1 and S2 of the `alloc` function. Although native execution calls `memory.grow` more frequently, its memory footprint is lower as depicted in Fig. 4. This is explained by the fact that heap expansions occur mostly at stage S1, where the heap growth policy of ensuring at least  $H\% := 33\%$  headroom in the heap applies. This definitely hurts WebAssembly as disclosed in Table IV. In addition to a higher memory under-utilization, a larger and highly fragmented heap (as shown in Table III) significantly raises the time to scan the heap, particularly when “finish marking” is triggered. Simply put, *this experiment confirms Observation 3, as well as reveals the high impact of a smaller mark-stack in the collector time.*

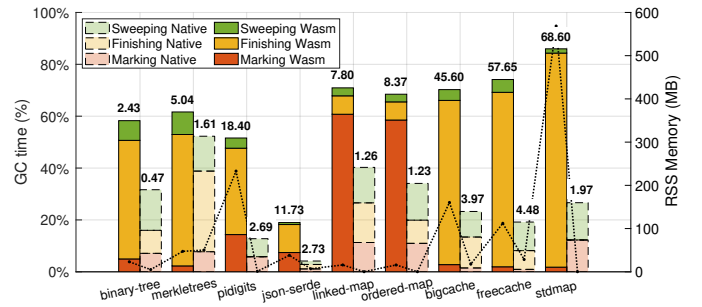


Fig. 4: **Performance comparison.** WebAssembly vs. native with the default configuration. Plotted line shows RSS values.

**Impact of latency-optimal configuration.** Here we study the performance-memory tradeoff when the default GC parameter values are adjusted to the latency-optimal values. Fig. 5 shows a significant reduction in garbage collection overhead in both cases. The blue squares above the bars list the latency-optimal configuration. The top value is the heap threshold ( $H\%$ ), and the bottom value is the mark-stack size ( $S$ ). *lru-linked-map*, for instance, reduces the garbage collection time to  $<10\%$ . In applications that require deep mark-stacks like *binary-tree* or *merkletrees*, it is even possible to entirely eliminate the “finishing marking” phase with a modest increase in the mark-stack size.

Nevertheless, the above gains are at the expense of memory. This is clearly visible in the heatmaps of Fig. 6. The heatmaps show that *merkletrees* can achieve speedups greater than 2x for WebAssembly, and only 1.2x for native, but at the cost of a 3-fold memory increase in WebAssembly vs. a 2-fold increase in native. Simple math suggests us that WebAssembly has more room for improvement:  $\frac{2}{3} > \frac{1.2}{2}$ , i.e., a better Pareto-front.

**Observation 4:** Default garbage collection configuration severely hurts WebAssembly performance. However, the adjustment of these parameters to the latency-optimized values yields an improvement superior to that of native. Due to an increased memory footprint, the performance-memory trade-off must be navigated with care to deliver improved performance but without exhausting memory.

**Parameter sensitivity.** A key question is then to determine the influence of each parameter on the garbage collection time. As shown in Fig. 6, these two parameters affect the performance-memory tradeoff differently. A higher heap threshold improves performance to a large extent, but also has the greatest impact on memory by fueling heap expansions. In contrast, the mark-stack size barely affects memory usage and shows diminishing returns on performance after reaching the optimal value:  $S_{OPT}$ . This is apparent in Table VI where we detail the coefficient of variation (CV) of latency and memory for all mark-stack sizes  $S$  s.t.  $S_{OPT} \leq S \leq 256$  (the maximum size not causing the mark-stack to escape to the heap). The lower CVs prove negligible impact on the latency-space tradeoff beyond the optimal mark-stack size. This suggests a simple “rule-of-thumb”: *To optimize latency, set  $S := 256$  and find the right heap threshold  $H\%$  for the application.*

TABLE VI: **Mark-stack size impact** on the latency-memory tradeoff. Coefficient of variation (CV) from the optimal mark-stack size  $S_{OPT}$  to maximum size (256). Default heap threshold.

Binary	$S_{OPT}$	Speedup	CV (Time)	CV (RSS)
<i>bigcache</i>	256	1.26x	0.0%	0.0%
<i>binary-tree</i>	96	1.45x	0.0%	0.26%
<i>freecache</i>	32	1.02x	16.05%	5.17%
<i>json-serde</i>	64	1.01x	0.48%	0.89%
<i>lru-linked-map</i>	224	1.12x	0.07%	0.14%
<i>lru-ordered-map</i>	160	1.12x	0.0%	0.49%
<i>merkle-trees</i>	160	1.71x	0.31%	0.15%
<i>pidigits</i>	128	1.05x	3.43%	2.92%
<i>stdmap</i>	96	1.36x	0.45%	0.02%

**Observation 5:** The heap threshold  $H\%$  dominates GC performance. The mark-stack size  $S$  has minor impact.

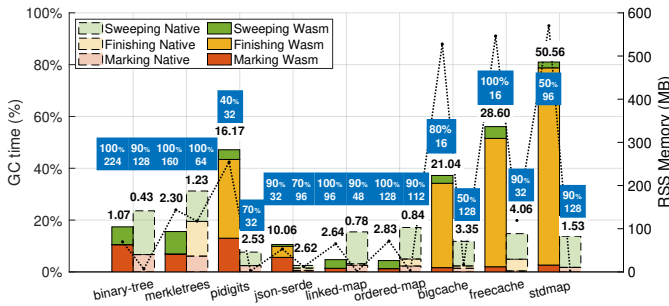


Fig. 5: **Performance improvement.** WebAssembly vs. native with latency-optimal configurations (blue squares above bars).

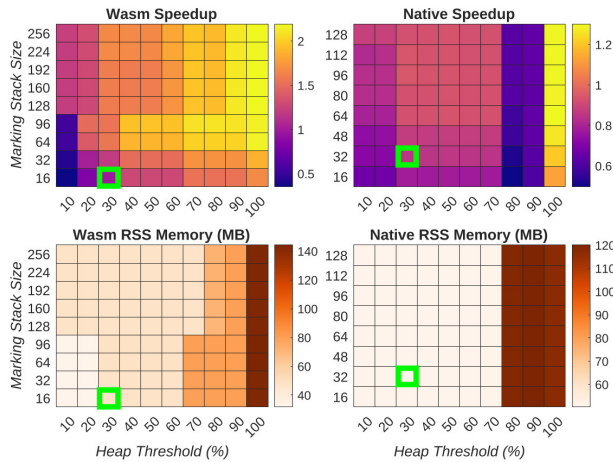


Fig. 6: **Performance-memory tradeoff.** Heatmaps capture the influence of the heap threshold vs. mark-stack size. The green boxes denote the default configuration.

## V. A CASE STUDY OF IMPROVED BATCH EXECUTION

### A. Overview

One important finding of our study is the existence of high variability across the latency-optimized configurations for each benchmark. Due to the dominance of the heap threshold  $H\%$ , this finding underscores the impracticality of adopting a “one-size-fits-all” threshold for all programs. Although this insight is not new [31], a fixed heap threshold is more harmful in WebAssembly than in native code as discussed above. If one

pursues to prove the feasibility of bringing multi-tenancy to the edge at comparable native performance, automatic fine-tuning of garbage collection in WebAssembly binaries is decisive.

As an example, we consider a scenario that often arises in practice: a batch scenario, where a batch of jobs are submitted to a batch scheduler almost at the same time [32]. The specific job selection policy is irrelevant to us and can be as simple as a ‘first-come-first-served’ (FCFS) policy. What we optimize here is the total **job completion time** (JCT), i.e., the time to finish the execution of the chosen jobs in the batch at one scheduling cycle. To do so, we automatically adjust their **heap thresholds** to trade off memory usage for garbage collection delays. In a nutshell, the basic idea is that the total time spent in garbage collection reduces in the right proportion in each job in order to minimize the total JCT.

We recall that determining the optimal heap threshold is a classic time-space trade-off problem [33]. Setting the threshold too low causes the GC to fire too often, pausing the application in an unacceptable manner. Setting it too high leads to memory exhaustion and high interference between processes. Although it is common practice to manually adjust the GC [34], manual tuning is impractical in the environments where WebAssembly makes perfect sense, e.g., a multi-tenant serverless edge system for running microservices with low startup times [5].

To be of practical use, we design our solution to fulfill the following properties:

- **P1–Black-box.** Our approach adjusts the heap thresholds from external observations alone, without any knowledge of the garbage-collected application executing within the WebAssembly VM (in our case, Wasmtime).
- **P2–Universal.** Our approach does not require modifying the underlying VM whatsoever, so each edge provider has the power to choose its most convenient runtime, either be Wasmtime or another. The same can be told for the batch scheduler, since our aim is to optimize the total JCT of the jobs selected by the scheduler at a given scheduling cycle.
- **P3–Uncoordinated.** Our framework does not require any communication between the VMs co-located in the same hardware. Each VM executes a given application without exchanging data with the rest of co-resident VMs to fine-tune its heap threshold. The absence of communication is particularly attractive in an untrusted environment like the edge, as no communication implies no side channels that could compromise WebAssembly SFI properties.

To meet the above properties, our solution basically requires an **online optimizer component** that tunes the heap thresholds of the selected jobs in the batch. This is achieved by solving a constrained optimization problem based on two types of black-box observations: JCTs and memory usage. The solution of the ILP formulation is the optimal set of heap thresholds. Once the thresholds are identified, we **binary-rewrite** the WebAssembly binaries to modify the default TinyGo threshold value of  $\frac{1}{3}$  to the optimal ones. We use binary-rewriting to ensure properties P1 and P2 hold true. On the one hand, binary-rewriting avoids the need to collect the source code and recompile it each time a new threshold is needed (**P1**), which is unrealistic for cloud-

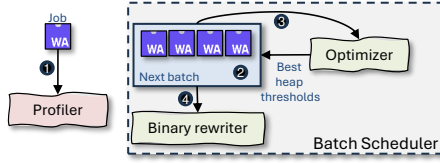


Fig. 7: A typical batch workflow.

TABLE VII: Source code recompilation vs. binary-rewriting for dynamic threshold replacement. Average of 100 runs.

Binary	Recompilation (s)		Binary-rewriting (s)	
	Raspberry	Mac M1	Raspberry	Mac M1
<i>bigcache</i>	18.27	2.34	1.51	0.53
<i>binary-tree</i>	8.52	1.21	0.66	0.37
<i>freecache</i>	21.18	2.55	1.37	0.50
<i>json-serde</i>	27.15	3.26	2.34	0.66
<i>lru-linked-map</i>	9.66	1.32	0.71	0.40
<i>lru-ordered-map</i>	8.61	1.23	0.70	0.39
<i>merkeltrees</i>	9.02	1.20	0.66	0.38
<i>pidigits</i>	22.04	2.70	1.56	0.53
<i>stdmap</i>	13.40	1.69	1.00	0.45

edge deployments. This model emulates that of AWS Lambda for compiled languages such as C++, Rust and Go, where the code is compiled natively by the user, and only the executable is uploaded to the cloud [14]. On the other hand, it avoids re-implementing the threshold replacement logic inside each VM of choice (P2). In most cases, our binary-rewriter takes under a second to rewrite a binary. If needed, the binary-rewriting of large binaries may be performed offline by rewriting the binary multiple types with a predefined set of thresholds. In Table VII, we report the time to binary-rewrite some of the benchmarks. As shown in the table, binary rewriting takes under 0.5 seconds in most of the binaries in the M1 MacBook Air, a low overhead that demonstrates the practicality of the approach.

To be useful, the optimizer needs to collect per-job samples for distinct heap thresholds. The way of collecting the samples is orthogonal to this work, since our purpose is to exploit the “malleability” of garbage collection to improve WebAssembly execution. We simply assume the existence of a job profiler.

**Workflow.** Fig. 7 illustrates a typical workflow of our solution. It consists of the following steps. ❶ The profiler executes the jobs offline and estimates their completion times and memory usage. ❷ After the profiling is completed, the batch scheduler is ready to dispatch the next batch of jobs. Once the scheduling decision is made, the batch scheduler calls the online optimizer to improve batch execution. ❸ Using profiling information, the optimizer uses an ILP solver to find the optimal per-job heap thresholds. ❹ To conclude, the batch scheduler binary rewrites the job binaries and puts them into execution.

### B. Analytical model

In practice, improving batch execution by playing out with the garbage collector becomes an optimization problem.

**Notations.** First, let  $[X]$  denote the ordered set  $\{1, 2, \dots, X\}$ . We consider  $K$  different thresholds for the remaining free heap space. Let  $\overline{\mathcal{H}} := \{\overline{H}_k : k \in [K]\}$  be the ordered set of these

thresholds such that for  $i, j \in [K]$  where  $i < j$ , then  $\overline{H}_i < \overline{H}_j$  (ascending order).

In our scenario, we consider  $n$  WebAssembly jobs  $J_i$ ,  $i \in [n]$ , are to be dispatched in a given edge server  $\ell$  with memory capacity  $M_\ell(t) \in \mathbb{N}_0$  at scheduling time  $t$ . The server has also  $n$  computation slots, each dedicated to run a separate job. That is, we assume a 1:1 mapping between jobs and computation slots. Each job  $J_i$  is accompanied by two profile vectors:

- Memory requirements:  $\mathbf{m}_i := (m_i^{\overline{H}_k} : k \in [K])$ , where each component  $m_i^{\overline{H}_k}$  denotes the peak memory usage of the job when the GC used  $\overline{H}_k$  to control heap growth.
- Job completion times:  $\mathbf{t}_i := (t_i^{\overline{H}_k} : k \in [K])$ , where  $t_i^{\overline{H}_k}$  denotes JCT of the job when the GC used  $\overline{H}_k$  to control heap expansions.

We recall here that memory utilization is typically inversely correlated with completion time. That is, while  $m_i^{\overline{H}_k} \leq m_i^{\overline{H}_j}$  for  $i < j$ , JCTs decrease with large thresholds:  $t_i^{\overline{H}_k} \geq t_i^{\overline{H}_j}$ .

**Problem formulation.** Our aim is to pick the “right” threshold for each job in order to minimize the overall completion time, subject to the server memory constraints. Formally, we have:

$$\text{minimize } T(x) = \sum_{i=1}^n \sum_{k=1}^K t_i^{\overline{H}_k} x_{ik} \quad (1)$$

subject to:

$$\sum_{i=1}^n \sum_{k=1}^K m_i^{\overline{H}_k} x_{ik} < M_\ell(t) \quad (2)$$

$$\sum_{k=1}^K x_{ik} = 1, \quad \forall i \in [n] \quad (3)$$

$$x_{ik} \in \{0, 1\}, \quad \forall i \in [n], \forall k \in [K]. \quad (4)$$

The variable  $x_{ik}$  is either 0, implying threshold  $\overline{H}_k$  for job  $i$  has not been chosen, or equal to 1 implying threshold  $\overline{H}_k$  has been picked for job  $i$ . Observe that constraint (3) ensures that only one threshold per job is chosen. Constraint (2) guarantees that adjusting the per-job GC does not exceed memory limits at scheduling time  $t$ . Finally, it is important to note that it is not mandatory to have job profiles for the  $K$  different thresholds. If a given job has only samples for a small subset of  $\overline{\mathcal{H}}$ , there will be less options for this particular job, making the solution suboptimal with respect to  $\overline{\mathcal{H}}$  (the set of threshold values).

**Theorem 1 (NP-hardness).** *Problem (1) is NP-hard.*

*Proof.* We build a polynomial-time reduction to (1) from the multi-choice knapsack (MCK) problem, a classic optimization problem which is known to be NP-hard. In the MCK problem, we are given objects split into several different classes referred to as  $N_i$ . Each object  $j$  of class  $i$  has a weight  $w_{ij}$  and a value parameter  $v_{ij}$ . The objective is to choose exactly one object from each class to maximize the total value subject to capacity constraint  $C$ :

$$\text{maximize } Z(x) = \sum_{i=1}^m \sum_{j \in N_i} v_{ij} x_{ij} \quad (5)$$



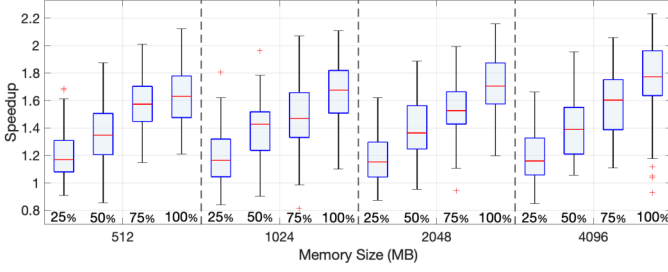


Fig. 8: **Speedup** of latency-optimized batching relative to non-optimized WebAssembly. Inlet percentages refer to  $p_{Q_1}$  %.

subject to the constraints: (6)  $\sum_{i=1}^m \sum_{j \in N_i} w_{ij} x_{ij} < C$ ; (7)  $\sum_{j \in N_i} x_{ij} = 1, \forall i \in [m]$ ; and (8)  $x_{ij} \in \{0, 1\}, \forall i \in [m], \forall j \in N_i$ . Given an instance  $A = (m, v_{ij}, w_{ij}, C, x_{ij})$  of the MCK problem, one can easily map it to an instance of problem (1) as  $A' = (n \leftarrow m, -t_i^H \leftarrow v_{ij}, m_i^H \leftarrow w_{ij}, M_\ell \leftarrow C, x_{ik} \leftarrow x_{ij})$ . Noticeably, this mapping problem can be solved in polynomial time. Hence, if there exists an algorithm that solves problem  $A'$ , it also solves the corresponding MCK problem. As a result, the MCK problem can be treated as a special case of (1). Given the NP-hardness of the MCK problem, (1) is NP-hard.  $\square$

### C. Implementation

We implement our proof-of-concept prototype with  $\sim 500$  lines of code. To implement our binary-rewriting mechanism, we utilize Walrus [35], a WebAssembly transformation library written in Rust. For the online optimizer, we leverage Google OR-Tools [36], a solver for hard optimization problems, and in particular, we use its SCIP solver to resolve problem (1) within millisecond range despite its NP-hardness (Theorem 1).

To not needlessly complicate the implementation, we build our batch scheduler from scratch. In brief, the batch scheduler is a FCFS scheduler with two job queues. One queue contains memory-bound jobs and the other CPU-bound tasks. The aim of the two queues is to enable us to quantify the gains of our optimization for mixed workloads. In a scheduling round, the scheduler creates a batch by performing independent Bernoulli trials with “success” probability  $p_{Q_1}$  of sampling from the first queue. If the trial fails, the job is taken from the second queue. Further, to replicate low-memory devices in the Raspberry Pi, the scheduler leverages the `cgroup` [37] controller for memory known as “memory `cgroup`” to limit the jobs in the batch to a smaller amount of memory  $M_\ell$ . The scheduler does not factor in CPU oversubscription each job runs in a separate core.

## VI. EVALUATION

To quantify how fine-tuning garbage collection can improve batch execution, we ask two main questions:

- How far-reaching are the performance benefits that heap threshold optimization can provide to batch execution?
- What is the accuracy of our solution?

**Methodology.** On our quad-core Raspberry Pi 4B, we simulate four low-memory environments with maximum memory sizes  $M_\ell \in \{0.5\text{GB}, 1\text{GB}, 2\text{GB}, 4\text{GB}\}$ . For each environment, we run a hundred batches of 4 mixed jobs, with each job running in a

TABLE VIII: **Errors comparison.** MSE, RMSE, and MAPE for prediction accuracy.

$(M_\ell, p_{Q_1})$	Avg.(Speedup)		MSE	RMSE	MAPE(%)
	Real	Forecast			
(0.5GB, 25%)	1.21	1.18	0.01	0.10	3.37
(0.5GB, 50%)	1.36	1.34	0.02	0.13	5.36
(0.5GB, 75%)	1.60	1.53	0.03	0.17	6.34
(0.5GB, 100%)	1.63	1.65	0.03	0.18	8.16
(1GB, 25%)	1.20	1.21	0.003	0.06	2.59
(1GB, 50%)	1.39	1.40	0.02	0.15	6.26
(1GB, 75%)	1.49	1.51	0.02	0.13	5.77
(1GB, 100%)	1.66	1.66	0.02	0.13	5.05
(2GB, 25%)	1.18	1.19	0.003	0.06	2.81
(2GB, 50%)	1.40	1.43	0.01	0.10	3.62
(2GB, 75%)	1.54	1.57	0.01	0.09	3.83
(2GB, 100%)	1.72	1.74	0.02	0.13	5.32
(4GB, 25%)	1.20	1.21	0.002	0.05	1.54
(4GB, 50%)	1.40	1.42	0.01	0.09	3.22
(4GB, 75%)	1.59	1.65	0.02	0.13	5.56
(4GB, 100%)	1.75	1.85	0.04	0.20	8.32

separate CPU core. Recall that the amount of memory-bound jobs is controlled by the sampling parameter  $p_{Q_1}$ . Pretend that  $p_{Q_1} = 0.25$ , then this means that on average, only 1 out of the 4 slots will be filled with a memory-bound task. The rest will be CPU-intensive jobs. Both types of jobs come from our two benchmarks. In all cases, we set the mark-stack size  $s$  to 256, following the “rule-of-thumb” given in §IV-E.

**Performance results.** Fig. 8 depicts boxplots of the speedup ratio between latency-optimized and unoptimized batching as a function of  $p_{Q_1}$ . Non-surprisingly, speedup gets better as the device memory increases due to the latency-memory tradeoff: the higher the heap threshold, the lesser the number of garbage collection cycles, so as the overhead. For  $p_{Q_1}=100\%$ , speedup reaches 2.22x for the 4GB setting. Notably, improvements are still evident even in the 0.5GB setting with speedups up to 1.7x in the worst case ( $p_{Q_1} = 25\%$ ). To complete the picture, Fig. 9 compares our latency-optimized method against native batched execution for the 4GB environment, as this is the one that has more headroom for improvement. The one bright note to raise here is that although WebAssembly somewhat still lags behind native performance, our optimization reduces latency to within 100% of the native JCT in all settings. Interestingly, this figure also shows that default, unoptimized WebAssembly is severely impacted by the  $p_{Q_1}$  value, i.e., whether the workload is more memory- or CPU-bound. Conversely, the optimized version is less sensitive and keeps a consistent JCT across all workloads.

**Observation 6:** Latency-optimized garbage collection is able to consistently maintain WebAssembly within 100% of the native JCT, regardless of the mixed workload type.

Finally, Fig. 10 shows the time taken by our online optimizer to solve problem (1) on two devices: MacBook M1 (blue bars), and Raspberry Pi (red bars), for an increasing batch size. Both subfigures convey a solving time below 0.01s on the MacBook and inferior to 0.2s on the Raspberry Pi 4 for all batch sizes. This proves the feasibility to run the optimizer inline with the edge batch scheduler.

**Accuracy results.** To conclude this paper, we investigate the accuracy of our latency-optimized method. Table VIII lists the

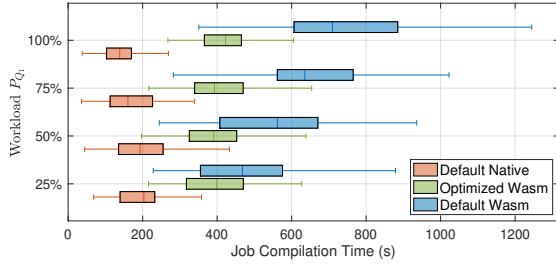


Fig. 9: **JCT comparison** as a function of  $p_{Q_1}$  (Y-axis) in the largest memory setting ( $M_\ell = 4\text{GB}$ ).

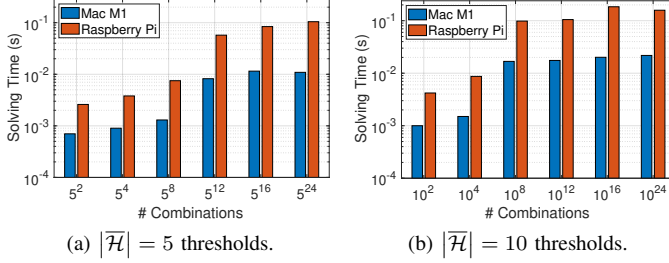


Fig. 10: **Time to solve optimization problem (1)** for batches of up to 24 applications.

existing error between the speedup predicted by the optimizer and the real speedup using three measures: Mean Square Error (MSE), Root Mean Square Error (RMSE) and Mean Absolute Percentage Error (MAPE). As shown in the table, error is small in all settings. Nevertheless, we observe a larger MAPE for the workloads where all tasks are memory-bound ( $p_{Q_1} = 100\%$ ). In the worst case, MAPE amounts to only 8.32%, which is not sufficiently significant to make our optimization impractical.

To complement the above result, we also wanted to examine whether or not the optimizer has a certain bias in the selection of heap thresholds. To minimize JCT, it is expected that there was some bias towards higher heap thresholds. To show this, Fig. 11 depicts the cumulative distribution function (CDF) of the chosen heap thresholds for two workloads: mixed workload with one single memory-bound job ( $p_{Q_1} = 25\%$ ), and the full memory-bound workload ( $p_{Q_1} = 100\%$ ). Irrespective of the workload, we find that more than 50% of the chosen thresholds are above the  $H\% := 60\% - 80\%$  range, which is much higher than the default threshold  $H\% := 33\%$ . We also notice a certain selection bias that depends on the amount of available memory  $M_\ell$ . For  $p_{Q_1} = 100\%$  (Fig. 11b), only 40% of the thresholds exceed  $H\% := 80\%$  to prevent running out of memory. As the amount of memory increases, thresholds become higher.

**Observation 7:** The optimizer is accurate and chooses thresholds much larger than  $H\% := 33\%$  to minimize the JCT yet maximizing memory usage.

## VII. RELATED WORK

With the breakthrough of WebAssembly, a number of works have put under close scrutiny the performance of this portable, low-level binary format from different angles [1], [10], [38]–[40]. But to the date, perhaps surprisingly, there is no study of the performance of WebAssembly for Go code, and even less

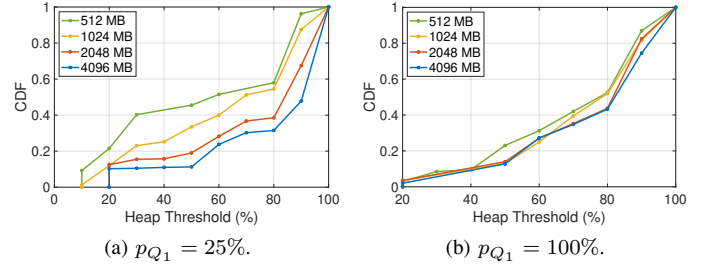


Fig. 11: **CDFs of heap thresholds** as calculated by the solver.

of garbage collection outside the browser. This is what makes this research such a valuable contribution to the community.

Outside WebAssembly, a series of studies have been carried out for STW collectors, generally aimed to improve execution time such as [41] for Java programs, where the authors suggest the sharp growth of the heap to reduce GC pauses if sufficient memory is available. Once again for Java, [42] proposed a PID controller that monitors GC overhead and adjusts the heap size accordingly to keep the GC overhead at a fixed level. Similarly, Tavakolisomah et al. [31] devised a new CPU-driven approach to dynamically fine-tune the heap size in concurrent collectors. Another work [43] proposed to exploit application idle times to hide expensive garbage collection operations. Lastly, Kirisame et al. [33] derived an optimal “square-root” heap limit rule to ensure that multiple heaps allocate memory among themselves in a manner that minimizes garbage collection time. Compared with our approach, all these solutions are “white-box” methods that require modifying (or instrumenting) the underlying Java and JavaScript virtual machines. Although a deeper application knowledge may yield a better trade-off between memory usage and garbage collection time, we opted for a solution that does not break the portability of WebAssembly across platforms and virtual machines.

## VIII. CONCLUSIONS

Although WebAssembly is compelling outside the browser, little is known of its performance for managed languages such as Golang. In this work, we have studied for the first time how WebAssembly virtualization perturbs ‘stop-the-world’ garbage collection for Go code. Our results report that garbage collection has a higher overhead in WebAssembly, but also that there is more headroom for improvement than native execution with proper tuning of the garbage collector. To show this, we have invented a new black-box approach to automatically minimize the total running time of a batch of jobs by adjusting the heap size thresholds as a tuning knob. Our evaluation demonstrates that our technique can reduce latency by up to 2x.

## ACKNOWLEDGMENT

This research has been partially funded by the EU under the Horizon Europe programme (CloudSkin, Grant 101092646), the Spanish MICIU/AEI Grant no. PID2023-148202OB-C21, and NextGenerationEU (CLOUDLESS UNICO I+D CLOUD 2022 project). Safia Guellil is a Martí Franquès Research Grant Fellow. Marc Sánchez-Artigas is a Serra Hùnter Fellow.

## REFERENCES

- [1] A. Haas et al., “Bringing the web up to speed with webassembly,” *SIGPLAN Not.*, vol. 52, no. 6, pp. 185–200, jun 2017. [Online]. Available: <https://doi.org/10.1145/3140587.3062363>
- [2] S. Hykes, “Wasm+wasi: An alternative to linux containers,” 2021, accessed: January 17, 2024. [Online]. Available: <https://twitter.com/solomonstre/status/1111004913222324225?lang=en>
- [3] M. Goedtel et al., “Create webassembly system interface (wasi) node pools in azure kubernetes service (aks) to run your webassembly (wasm) workload (preview),” 2023, accessed: February 3, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/aks/use-wasi-node-pools>
- [4] “Kwasm – kubernetes operator for webassembly,” 2023, accessed: February 5, 2024. [Online]. Available: <https://kwasm.sh/>
- [5] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, “Sledge: a serverless-first, light-weight wasm runtime for the edge,” in *21st International Middleware Conference (Middleware’20)*, 2020, pp. 265–279. [Online]. Available: <https://doi.org/10.1145/3423211.3425680>
- [6] B. Li, W. Dong, and Y. Gao, “Wipro: A webassembly-based approach to integrated iot programming,” in *IEEE Conference on Computer Communications (IEEE INFOCOM 2021)*, 2021, pp. 1–10.
- [7] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, “Pushing serverless to the edge with webassembly runtimes,” in *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid’22)*, 2022, pp. 140–149.
- [8] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, “Webassembly as a common layer for the cloud-edge continuum,” in *2nd Workshop on Flexible Resource and Application Management on the Edge (FRAME’22)*, 2022, pp. 3–8. [Online]. Available: <https://doi.org/10.1145/3526059.3533618>
- [9] S. Shillaker and P. Pietzuch, “Faasm: Lightweight isolation for efficient stateful serverless computing,” in *2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC’20)*, USA, 2020.
- [10] D. Lehmann, J. Kinder, and M. Pradel, “Everything old is new again: Binary security of webassembly,” in *29th USENIX Conference on Security Symposium (SEC’20)*, 2020.
- [11] A. Zakai, “A new way to bring garbage collected programming languages efficiently to WebAssembly,” 11 2023. [Online]. Available: <https://v8.dev/blog/wasm-gc-porting>
- [12] Tinygo-Org, “wasm: switch to wasi\_snapshot\_preview1 by aykevl · Pull Request #1690 · tinygo-org/tinygo,” 2021. [Online]. Available: <https://github.com/tinygo-org/tinygo/pull/1690>
- [13] ForgeRock, “Best practice for jvm tuning with g1 gc,” 2024, accessed: May 22, 2024. [Online]. Available: <https://backstage.forgerock.com/knowledge/kb/article/a75965340>
- [14] —, “When to use lambda’s os-only runtimes,” 2024, accessed: August 2, 2024. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/runtimes-provided.html>
- [15] B. Alliance, “Wasmtime – a standalone runtime for webassembly,” 2023. [Online]. Available: <https://github.com/bytecodealliance/wasmtime>
- [16] F. Denis, “Memory management in webassembly: guide for c and rust programmers,” 2019. [Online]. Available: <https://www.fastly.com/blog/webassembly-memory-management-guide-for-c-rust-programmers>
- [17] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, pp. 203–216, dec 1993. [Online]. Available: <https://doi.org/10.1145/173668.168635>
- [18] “The llvm compiler infrastructure,” 2023. [Online]. Available: <https://github.com/llvm/llvm-project>
- [19] “Wasi – the webassembly system interface,” 2023. [Online]. Available: <https://wasi.dev/>
- [20] “wasi-libc,” 2024. [Online]. Available: <https://github.com/WebAssembly/wasi-libc>
- [21] “Benchmarks for programming languages and compilers, Which programming language or compiler is faster.” [Online]. Available: <https://programming-language-benchmarks.vercel.app/>
- [22] R. Garner, S. M. Blackburn, and D. Frampton, “Effective prefetch for mark-sweep garbage collection,” in *6th International Symposium on Memory Management (ISMM ’07)*, 2007, pp. 43–54. [Online]. Available: <https://doi.org/10.1145/1296907.1296915>
- [23] H. Onozawa, T. Ugawa, and H. Iwasaki, “Fusuma: double-ended threaded compaction,” in *2021 ACM SIGPLAN International Symposium on Memory Management (ISMM 2021)*, 2021, pp. 94–106. [Online]. Available: <https://doi.org/10.1145/3459898.3463903>
- [24] B. Alliance, “Cranelfit code generator,” 2023. [Online]. Available: <https://github.com/bytecodealliance/wasmtime/tree/main/cranelfit>
- [25] H. Xu and F. Kjolstad, “Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode,” *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: <https://doi.org/10.1145/3485513>
- [26] “Benchmarks for bigcache project,” 2024, accessed: February 8, 2024. [Online]. Available: <https://github.com/allegro/bigcache-bench>
- [27] C. Bays, “A comparison of next-fit, first-fit, and best-fit,” *Commun. ACM*, vol. 20, no. 3, pp. 191–192, Mar. 1977. [Online]. Available: <https://doi.org/10.1145/359436.359453>
- [28] R. Jones and R. Lins, *Garbage collection: algorithms for automatic dynamic memory management*, USA, 1996.
- [29] Abhinav and R. Nasre, “Fastcollect: offloading generational garbage collection to integrated gpus,” in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES ’16)*, 2016. [Online]. Available: <https://doi.org/10.1145/2968455.2968520>
- [30] N. Cohen and E. Petrank, “Data structure aware garbage collector,” *SIGPLAN Not.*, vol. 50, no. 11, pp. 28–40, Jun. 2015. [Online]. Available: <https://doi.org/10.1145/2887746.2754176>
- [31] S. Tavakolisomeh, M. Shimchenko, E. Österlund, R. Bruno, P. Ferreira, and T. Wrigstad, “Heap size adjustment with cpu control,” in *20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2023)*, 2023, pp. 114–128. [Online]. Available: <https://doi.org/10.1145/3617651.3622988>
- [32] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, “Network-aware scheduling for data-parallel jobs: Plan when you can,” in *2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM ’15)*, New York, NY, USA, 2015, pp. 407–420. [Online]. Available: <https://doi.org/10.1145/2785956.2787488>
- [33] M. Kirisame, P. Shenoy, and P. Panchekha, “Optimal heap limits for reducing browser memory use,” *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, Oct. 2022. [Online]. Available: <https://doi.org/10.1145/3563323>
- [34] X. Shang, Y. Zhang, F. Li, A. Sampath, and G. Baliga, “Tricks of the trade: Tuning jvm memory for large-scale services,” 2020, accessed: September 11, 2024. [Online]. Available: <https://eng.uber.com/jvm-tuning-garbage-collection/>
- [35] T. Rust and W. W. Group, “Walrus,” 2024, accessed: January 14, 2024. [Online]. Available: <https://github.com/rustwasm/walrus>
- [36] Google, “Google or-tools,” 2024, accessed: July 17, 2024. [Online]. Available: <https://developers.google.com/optimization/>
- [37] T. Heo, “Control group, v2,” 2015, accessed: June 18, 2024. [Online]. Available: <https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html>
- [38] 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, Jul. 2019, pp. 107–120. [Online]. Available: <https://www.usenix.org/conference/atc19/presentation/jangda>
- [39] Y. Zhang et al., “Characterizing and detecting webassembly runtime bugs,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, Dec. 2023. [Online]. Available: <https://doi.org/10.1145/3624743>
- [40] Z. Liu, D. Xiao, Z. Li, S. Wang, and W. Meng, “Exploring missed optimizations in webassembly optimizers,” in *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’23)*, 2023, pp. 436–448. [Online]. Available: <https://doi.org/10.1145/3597926.3598068>
- [41] T. Brecht, E. Arjomandi, C. Li, and H. Pham, “Controlling garbage collection and heap growth to reduce the execution time of java applications,” in *16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’01)*, 2001, pp. 353–366. [Online]. Available: <https://doi.org/10.1145/504282.504308>
- [42] D. R. White, J. Singer, J. M. Aitken, and R. E. Jones, “Control theory for principled heap sizing,” in *Proceedings of the 2013 International Symposium on Memory Management (ISMM ’13)*, 2013, pp. 27–38. [Online]. Available: <https://doi.org/10.1145/2491894.2466481>
- [43] U. Degenbaev, J. Eisinger, M. Ernst, R. McIlroy, and H. Payer, “Idle time garbage collection scheduling,” in *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’16)*, 2016, pp. 570–583. [Online]. Available: <https://doi.org/10.1145/2908080.2908106>