Taras Shevchenko National University of Kyiv
Faculty of Computer Science and Cybernetics
Department of Intelligent Software Systems

THESIS

Bulk memory operations implementation in WebAssembly

Submitted as Fulfilment of the Requirements for the Degree of M.Sc. in Software Engineering

Author: 2nd year masters student
Oleksandr R. Syrotenko

_____

Academic Supervisor:
PhD in Engineering, Assistant Professor
Ievgen O. Demkivskiy

_____

I certify that this thesis does not contain any borrowings from the works of other authors without corresponding references. Author _____

The thesis was considered and reccomended for defence on the session of Department of Intelligent Software Systems, protocol No. 11 from May 13, 2019.

Head of the Department
Professor O.I. Provotar _____

Kyiv — 2019

Abstract

Bulk memory operations is a common memory calculation optimization technique that takes some memory pointer and make massive store to this memory. A lot of programs have intensive computations that use bulk operations like `memcpy` or `memmove` to improve copying or storing big chunks of memory. There are a lot of places where programmer can improve memory consumption and computation complexity of them application by increasing througput via bulk operations usage.

This thesis identifies bulk memory operations concept and implementation in WebAssembly virtual machine. It also presents approaches of optimtization connected with bulk operations : conditional segment initialization and passive segment initialiation. Bulk operations have their own intrinsics and any programmer will have a possibility to use them in their code. Additionally, any language-to-WebAssembly compiler can map its internal lanuguage bulk operations to a new WebAssembly bulk memory operations.

All of the algorithms have been implemented in the Microsoft ChakraCore engine and tested on two devices : MacBook Pro 2015 with Retina equipped with a x86-64 processor (Haswell) and PC with Intel Core i5-7600 x86-64 processor and 16 GB of RAM using internal ChakraCore test suite, custom tests and benchmarks.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

There are near 4 billion Internet users in the world today. The very big part of them are using browsers for surfing the Internet. It makes browsers the most popular software application in the world (except operating systems) and it means that overall time of code execution and web-page loading should be minimized. There is an investigation of 'USA Today' which shows that :

— Amazon could lose 1.6 billion dollars per year if their web-page delay increase by 1 second;

— Slowing Google's search result by 4/10's of second would reduce number of searches by 8 million in a day;

— One of four web-site visitors would abandon this page if it takes more than a second to load it;

— One of five web-site visitors become rude to web-service because they 'serving them too slowly';

All these facts mean one thing - it's profitable both for browser manufacturers(which are the biggest serivces providers) and for developers community to invest in developing and improving the productivity of websites and web applications.

WebAssembly is of the major result of mutual research and design of performant Web tools. WebAssembly was was first announced 17th of June 2015, and the first demonstration was executing Angry Bots in three major browsers. The precursor technologies were asm.js from Mozilla and Google Native Client (PNaCl) and the initial implementation was based on the feature set of `asm.js`. [1]

WebAssembly (as a rule, this term is shortened to WASM) is a standard that defines a portable binary format and a corresponding assembly-like text format for executables. The main goal of the format is to enable high performance

applications on web pages, but it is designed to be executed and integrated in other environments as well, not only in Web-applications. [1]

Original memory management in WebAssembly.

Sometimes software engineers want to copy or initialize big chunk of linear memory.

For this reason they are using `memset/memcpy` in C/C++ or `System.arraycopy()` in Java, generally called as **bulk memory operations**.

```
#include <string.h>
int testFunction() {
  size_t length = 1024;           // 2^10 for test purposes
  int *array1 = (int*) malloc(length * sizeof(array1));
  // ... some usage of this mempory here
  int *array2 = (int*) malloc(length * sizeof(array2));

  for(int i = 0; i < length; ++i)
    array1[i]++;                  // avoid constant propagation.
  memcpy(array2, array1, length); // copy data from array1 to array2
  memset(array1, 0, len);         // filling all array1 memory by zeroes
  return array1[0] + array2[0];   // avoid dead code elimination.
}
```

Compilation result of this fucntion by `emscripten` tool to such piece of code (full code is available in Appendix A) :

```
(module
 (import "env" "memset" (func $memset (param i32 i32 i32) (result i32)))
 (import "env" "memcpy" (func $memcpy (param i32 i32 i32) (result i32)))
 (table 0 anyfunc)
 (export "memory" (memory $0))
 (export "_Z12testFunctionPi" (func $_Z12testFunctionPi))

 // compiled testFunction.
 (func $_Z12testFunctionPi (; 3 ;) (param $0 i32) (result i32)
  (local $1 i32)  (local $2 i32) (local $3 i32) (local $4 i32)
```

```
  (set_local $1
   (call $memset          // <- memset call
    (call $_Znaj
     (i32.const 4096)
    )
    (i32.const 10)
    (i32.const 1024)
   )
  )
  // ... a lot of generated code with folded constants.
  (i32.load
   (call $memcpy          // <- memcpy call
    (get_local $2)
    (get_local $1)
    (i32.const 1024)
   )
  )
 ))
)
```

Compiled code contains `memcpy` and `memset` functions. They were imported from `env` module which ***is linked*** with standart `libc` library, where those functions are defined and implemented. It would be better to use internal WebAssembly bulk operations instead of using `libc`'s memory operations.

This is a key idea of further investigations : desiging and implementation of bulk memory operations intrinsics, which would be placed to a special `memory` module.

Summary of Major Contributions.

The major contributions of this thesis are:

— Description of algorithms of bulk copy, move and initialization operations on WebAssembly memory.

— Development of passive memory initialization algorithms.

— Implementation of bulk operations algorithms.

— Integrations of algorithms to a production virtual machine named Chakra-

Core.

— Study and comparison of the performance of these algorithms compare to standard.

The algorithms have been implemented in C++ and integrated to the ChakraCore engine. They can be used as comilation result of any compiled user's WebAssembly module. These algorithms are fully compatible with WebAssembly specification and bulk memory operations proposal.

Chapter 1 provides introduction about WebAssembly as technology and its ancestors. Chapter 2 describes the major WebAssembly compilers and runtimes. Chapter 3 presents the developed algorithms for every bulk operation and their profitable usage. The results of compilation, runtime and benchmarks tests are presented in Chapter 4. Appendix A contains an example of WebAssembly S-expressions.

DEFINITIONS

— **SIMD** is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Such machines exploit **data level parallelism, but not concurrency**: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment.

CHAPTER 1
WEBASSEMBLY

1.1. Abstract

Nowadays browsers are the most popular application not only for surfing the Internet. There are a lot of office services (like word processors, spreadsheet editors, etc), games, video and audio streams, 3D editors are available in a browser; all of listed services require a good client performance for correct and fast work. Unfortunately, browsers cannot give such performance for this class of applications permanently. Every web-application could be written as :

— typical JavaScript + HTML + CSS web-page with some features. JavaScript has well-optimized just-in-time compiler with good startup and compilation time in every browser, but it's still not enough for maximum performance and safety.

— so-called 'rich' application, which requires special browser plugin and special applications for creating them. As a rule, they are executing in some external virutal machine. It means that application has middle-valued throughput and big memory footprint, which could be minimized.

Performance challenge produced start of design and implementation of new low-level browser formal called WebAssembly.

1.2. WebAssembly Overview

[2]

1.2.1. General.

WebAssembly (abbreviated WASM) is a safe, portable, low-level code format designed for efficient execution and compact representation. Its main goal is to

enable high performance applications on the Web, but it does not make any Web-specific assumptions or provide Web-specific features, so it can be employed in other environments as well.

WebAssembly is an open standard developed by a W3C Community Group.

1.2.2. Design Goals.

The design goals of WebAssembly are the following:

— **Fast**: executes with near native code performance, taking advantage of capabilities common to all contemporary hardware.

— **Safe**: code is validated and executes in a memory-safe, sandboxed environment preventing data corruption or security breaches.

— **Well-defined**: fully and precisely defines valid programs and their behavior in a way that is easy to reason about informally and formally.

— **Hardware-independent**: can be compiled on all modern architectures, desktop or mobile devices and embedded systems alike.

— **Language-independent**: does not privilege any particular language, programming model, or object model.

— **Platform-independent**: can be embedded in browsers, run as a standalone VM, or integrated in other environments.

— **Open**: programs can interoperate with their environment in a simple and universal manner.

— **Compact**: has a binary format that is fast to transmit by being smaller than typical text or native code formats.

— **Modular**: programs can be split up in smaller parts that can be transmitted, cached, and consumed separately.

— **Efficient**: can be decoded, validated, and compiled in a fast single pass, equally with either just-in-time (JIT) or ahead-of-time (AOT) compilation.

— **Streamable**: allows decoding, validation, and compilation to begin as soon as possible, before all data has been seen.

— **Parallelizable**: allows decoding, validation, and compilation to be split into many independent parallel tasks.

— **Portable**: makes no architectural assumptions that are not broadly supported across modern hardware. WebAssembly code is also intended to be easy to inspect and debug, especially in environments like web browsers, but such features are beyond the scope of this specification.

At its core, WebAssembly is a virtual instruction set architecture (virtual ISA). As such, it has many use cases and can be embedded in many different environments.

1.2.3. Concepts. [2]

WebAssembly encodes a low-level, assembly-like programming language. This language is structured around the following concepts:

— **Values**: WebAssembly provides only four basic value types. These are integers and IEEE 754-2008 numbers, each in 32 and 64 bit width. 32 bit integers also serve as Booleans and as memory addresses. The usual operations on these types are available, including the full matrix of conversions between them. There is no distinction between signed and unsigned integer types. Instead, integers are interpreted by respective operations as either unsigned or signed in two's complement representation.

— **Instructions**: the computational model of WebAssembly is based on a stack machine. Code consists of sequences of instructions that are executed in order. Instructions manipulate values on an implicit operand stack * and fall into two main categories. Simple instructions perform basic operations on data. They pop arguments from the operand stack and push results back to it. Control instructions alter control flow. Control flow is structured, meaning it is expressed with well-nested constructs such as blocks, loops, and conditionals. Branches can only target such constructs.

— **Traps**: under some conditions, certain instructions may produce a trap,

which immediately aborts execution. Traps cannot be handled by WebAssembly code, but are reported to the outside environment, where they typically can be caught.

— **Functions**: code is organized into separate functions. Each function takes a sequence of values as parameters and returns a sequence of values as results.** Functions can call each other, including recursively, resulting in an implicit call stack that cannot be accessed directly. Functions may also declare mutable local variables that are usable as virtual registers.

— **Tables**: a table is an array of opaque values of a particular element type. It allows programs to select such values indirectly through a dynamic index operand. Currently, the only available element type is an untyped function reference. Thereby, a program can call functions indirectly through a dynamic index into a table. For example, this allows emulating function pointers by way of table indices.

— **Linear Memory**: a linear memory is a contiguous, mutable array of raw bytes. Such a memory is created with an initial size but can be grown dynamically. A program can load and store values from/to a linear memory at any byte address (including unaligned). Integer loads and stores can specify a storage size which is smaller than the size of the respective value type. A trap occurs if an access is not within the bounds of the current memory size.

— **Modules**: a WebAssembly binary takes the form of a module that contains definitions for functions, tables, and linear memories, as well as mutable or immutable global variables. Definitions can also be imported, specifying a module/name pair and a suitable type. Each definition can optionally be exported under one or more names. In addition to definitions, modules can define initialization data for their memories or tables that takes the form of segments copied to given offsets. They can also define a start function that is automatically executed.

— **Embedder**: a WebAssembly implementation will typically be embedded

into a host environment. This environment defines how loading of modules is initiated, how imports are provided (including host-side definitions), and how exports can be accessed. However, the details of any particular embedding are beyond the scope of this specification, and will instead be provided by complementary, environment-specific API definitions.

* In practice, implementations need not maintain an actual operand stack. Instead, the stack can be viewed as a set of anonymous registers that are implicitly referenced by instructions. The type system ensures that the stack height, and thus any referenced register, is always known statically.

** In the current version of WebAssembly, there may be at most one result value.

1.2.4. Semantic Phases.

Conceptually, the semantics of WebAssembly is divided into three phases. For each part of the language, the specification specifies each of them.

Decoding. WebAssembly modules are distributed in a binary format. Decoding processes that format and converts it into an internal representation of a module. In this specification, this representation is modelled by abstract syntax, but a real implementation could compile directly to machine code instead.

Validation. A decoded module has to be valid. Validation checks a number of well-formedness conditions to guarantee that the module is meaningful and safe. In particular, it performs type checking of functions and the instruction sequences in their bodies, ensuring for example that the operand stack is used consistently.

Execution. Finally, a valid module can be executed. Execution can be further divided into two phases:

Instantiation. A module instance is the dynamic representation of a module, complete with its own state and execution stack. Instantiation executes the module body itself, given definitions for all its imports. It initializes globals, memories and tables and invokes the module's start function if defined. It returns the

instances of the module's exports.

Invocation. Once instantiated, further WebAssembly computations can be initiated by invoking an exported function on a module instance. Given the required arguments, that executes the respective function and returns its results.

Instantiation and invocation are operations within the embedding environment.

1.2.5. Enhancement process. WebAssembly community has big growth and they actively help to improve this technology. There are two divided community groups : [3] [4]

— **Working Group.** The scope of the WebAssembly Working Group comprises addressing the need for native-performance code on the Web in applications ranging from 3D games to speech recognition to codecs—and in any other contexts in which a common mechanism for enabling high-performance code is relevant—by providing a standardized portable, size-, and load-time-efficient format and execution environment that attempts to maximize performance and interoperate gracefully with JavaScript and the Web, while ensuring security and consistent behavior across a variety of implementations.

— **Community Group.** The mission of Community Group is to provide a forum for pre-standardization collaboration on the WebAssembly format. The Community Group is explicitly not intended to standardize the final ratified text of standards related to WebAssembly. Instead it will prepare recommendations to a separate WebAssembly Working Group.

There are a lot of community proposals inside the WebAssembly enhancement process. Let's count the most important proposals :

— **Reference types**. WebAssembly still doesn't include reference types. The key idea of this proposal is to support `nullref` and give a possibility to manipulate a tables inside WebAssembly memory.

— **Garbage collection**. WebAssembly doesn't have automatic memory

management system, which can be build after implementation of reference types. It gives a possibility to count references or trace whole 'heap', staring from the current rootset.

— **Bulk memory operations**. Fast big chunks of memory storing in different implementations.

— **Tail calls**. This proposal allows the correct and efficient implementations of languages that require tail call elimination.

— **Atomics and threads**. This proposal adds a new shared linear memory type and some new operations for atomic memory access. The responsibility of creating and joining threads is deferred to the embedder.

— **SIMD operations**. There are a lot of processors which have special registers for vectors computation (or SIMD, which descripted as single instruction, multiply data). All the most popular architectures have SIMD extensions. For example, Intel has AVX and SSE registers with 128 to 512 bytes capacity. Arm has NEON, PowerPC have AltiVEC and SPE, MIPS have MaDMaX technologies for SIMD operations. Of course, it is a great challenge to support SIMD operations both for compilers and for runtime, because it can improve the performanc of some complex computations in a times.

1.3. Overview of Rivals

1.3.1. Java Applets.

A Java applet is a small application is written in the Java language, or another programming language that compiles to Java bytecode (JVM intermediate representation), and delivered to users in the form of Java bytecode. The browser user launched the Java applet from a web page, and the applet was then executed within JVM were launched in separate process from the web browser itself. Programmers could appear a Java applets in any frame of the web page, a new application window, Sun's AppletViewer, or an another stand-alone tool.

Java applets had access to hardware acceleration, making them well-suited for non-trivial, computation-intensive visualizations. When applets were popular, JavaScript engines didn't have GPU support. As browsers have gained support for hardware-accelerated graphics thanks to the canvas technology (or specifically WebGL in the case of 3D graphics), as well as just-in-time compiled JavaScript became very efficient, the speed difference has become less noticeable.

The applet can be displayed on the web page by making use of the deprecated applet HTML element, or the recommended object element. The embed element can be used with Mozilla family browsers (embed was deprecated in HTML 4 but is included in HTML 5). This specifies the applet's source and location. Both object and embed tags can also download and install Java virtual machine (if required) or at least lead to the plugin page. `applet` and `object` tags also support loading of the serialized applets that start in some particular (rather than initial) state. Tags also specify the message that shows up in place of the applet if the browser cannot run it due to any reason [5].

Java applets were deprecated since JDK 9 in 2017 and removed from JDK 11, released in September 2018.

By definition, Java bytecode is platform independent intermediate representation, Java applets can be executed by browsers (or other clients) for many platforms, including Microsoft Windows, FreeBSD, Unix, macOS and Linux. They cannot be run on modern mobile devices, which do not support Java.

1.3.2. Microsoft Silverlight. [6]

Microsoft Silverlight (or simply Silverlight) was an application framework for writing and running rich Internet applications, similar to Adobe Flash. A plugin for Silverlight is still available for some browsers. While early versions of Silverlight focused on streaming media, later versions supported multimedia, graphics, and animation and gave developers support for CLI languages and development tools.

Silverlight provides a retained mode graphics system similar to Windows Presentation Foundation (WPF), and integrates multimedia, graphics, animations, and interactivity into a single run-time environment. In Silverlight applications, user interfaces are declared in Extensible Application Markup Language (XAML) and programmed using a subset of the .NET Framework. XAML can be used for marking up the vector graphics and animations.

Silverlight supports H.264 video, Advanced Audio Coding, Windows Media Video (WMV), Windows Media Audio (WMA), and MPEG Layer III (MP3) media content across all supported browsers without requiring Windows Media Player, the Windows Media Player ActiveX control, or Windows Media browser plug-ins. Silverlight exposes a Downloader object which can be used to download content, like scripts, media assets, or other data, as may be required by the application. With version II, the programming logic can be written in any .NET language, including some derivatives of common dynamic programming languages like **IronRuby** and **IronPython**.

Cross platform Mozilla Firefox support for Silverlight was removed in Firefox 52 released in March 2017 when Mozilla removed support for NPAPI plugins,[40] bringing it in-line with the removal of NPAPI plugin support in Google Chrome.

1.3.3. Adobe Flash. [7]

Adobe Flash Player is computer software for using content created on the Adobe Flash platform, including viewing multimedia contents, executing rich Internet applications, and streaming audio and video. Flash Player can run from a web browser as a browser plug-in or on supported mobile devices. Flash Player was created by Macromedia and has been developed and distributed by Adobe Systems since Adobe acquired Macromedia in 2005.

Flash Player runs **SWF** files that can be created by Adobe Flash Professional, Adobe Flash Builder or by third party tools such as FlashDevelop. Flash Player supports vector graphics, 3D graphics, embedded audio, video and raster graphics, and a scripting language called **ActionScript**. ActionScript is based

on ECMAScript (similar to JavaScript) and supports object-oriented code. Flash Player is distributed free of charge and its plug-in versions are available for every major web browser and operating system. Google Chrome, Internet Explorer 11 in Windows 8 and later, and Microsoft Edge come bundled with a sandboxed Adobe Flash plug-in.

Flash Player once had a large user base, and was a common format for web games, animations, and graphical user interface (GUI) elements embedded in web pages. Adobe stated in 2013 that more than 400 million out of over 1 billion connected desktops update to the new version of Flash Player within six weeks of release.

Flash Player has become increasingly criticized for its performance, consumption of battery on mobile devices, the number of security vulnerabilities that had been discovered in the software, and its closed platform nature.

In July 2017, Adobe announced that it would end support for Flash Player in 2020, and continued to encourage the use of open HTML5 standards in place of Flash. The announcement was coordinated with all tech giantes. Its usage has also waned because of modern web standards that allow some of Flash's use cases to be fulfilled without third-party plugins.

1.3.4. asm.js. [8]

`asm.js` a very small, strict subset of JavaScript that only allows things like `while`, `if`, numbers, top-level named functions, and other simple constructs. It does not allow objects, strings, closures, and basically anything that requires heap allocation. Asm.js code resembles C in many ways, but it's still completely valid JavaScript that will run in all current engines. It pushes JS engines to optimize this kind of code, and gives compilers like Emscripten a clear definition of what kind of code to generate. We will show what `asm.js` code looks like and explain how it helps and how you can use it.

This subset of JavaScript is already highly optimized in many JavaScript engines using fancy Just-In-Time (JIT) compiling techniques. However, by defining

an explicit standard we can work on optimizing this kind of code even more and getting as much performance as we can out of it. It makes it easier to collaborate across multiple JS engines because it's easy to talk about and benchmark. The idea is that this kind of code should run very fast in each engine, and if it doesn't, it's a bug and there's a clear spec that engines should optimize for.

It also makes it easy for people writing compilers that want to generate high-performant code on the web. They can consult the `asm.js` spec and know that it will run fast if they adhere to `asm.js` patterns. Emscripten, a C/C++ to JavaScript compiler, emits `asm.js` code to make it run with near native performance on several browsers.

Additionally, if an engine chooses to specially recognize asm.js code, there even more optimizations that can be made. Firefox is the only browser to do this right now.

1.3.5. Comparison WebAssembly with Rivals.

1.3.5.1. WebAssembly vs rich applications.

So-called 'rich' application is an application that is executing on browser plugin which supports programming language, unlike WebAssembly. WebAssembly application user doesn't require any additional plugins, this is native browser technology.

We can summarize all advantages in list of WebAssembly's major features which don't available in 'rich' applications :

— **Performance**. Designed as low-level binary instruction set, it provides high speed of parsing and execution which is higher than execution of 'rich' application. Also, it uses SIMD instructions, atomic operations. Threads and GC are intended to implement in WASM engines.

— **Security**. WebAssembly's applications and JS in browsers use same API for communication with hardware and OS.

— **JS compatibility with WebAssembly**.

— **Language-independent**. It does not privilege any particular language,

programming model, or object model, unlike in 'rich' applications, which depend from concrete target virtual machine or programming language.

— **Compact**. WebAssembly has a binary format which is smaller than typical text or native code formats.

— **Open**. WebAssembly is open format with community-driven specification, unlike 'rich' applications which have vendor-locked specification.

1.3.5.2. WebAssembly vs asm.js.

As it was defined higher, `asm.js` is just a subset of JavaScript language, which gives a possibility to write more performant code using types and low-level data manipulations. I assume that `asm.js`'s has that biggest problems: highly increased code complexity (complex developer experience) and small browser support (Firefox).

We can summarize all advantages of WebAssembly over `asm.js` [9]:

1. Startup.

WebAssembly is designed to be small to download and fast to parse, so that even large applications start up quickly.

It's actually not that easy to improve on the download size of gzipped minified JavaScript, as it's already fairly compact when compared with native code. Still, WebAssembly's binary format can improve on that, by being carefully designed for size in mind (indexes are LEB128s, etc.). It is often around 10–20 percent smaller (comparing gzipped sizes).

WebAssembly improves on parsing in a much bigger way: It can be parsed an order of magnitude faster than JavaScript. This mostly comes down to binary formats being faster to parse, especially ones designed for that. WebAssembly also makes it easy to parse (and optimize) functions in parallel, which helps a lot on multicore machines.

Total startup time can include factors other than downloading and parsing, such as the VM fully optimizing the code, or downloading additional data files that are necessary before execution, etc. But downloading and parsing are unavoidable

and therefore important to improve upon as much as possible. All the rest can be optimized or mitigated, either in the browser or in the app (for example, fully optimizing the code can be avoided by using a baseline compiler or interpreter for WebAssembly, for the first few frames).

2. CPU features.

One trick that's made `asm.js` so fast is that while all JavaScript numbers are doubles, in `asm.js` an addition will have a bitwise-and operation right after it, which makes it logically equivalent to the CPU doing a simple integer addition, which CPUs are very good at. So `asm.js` made it easy for VMs to use a lot of the full power of CPUs.

But `asm.js` was limited to things that are expressible in JavaScript. WebAssembly isn't limited in that way, and lets us use even more CPU features, such as:

— **64-bit integers**. Operations on them can be up to 4x faster. This can speed up hashing and encryption algorithms, for example.

— **Load and store offsets**. This helps very broadly, basically anything that uses memory objects with fields at fixed offsets (C structs, etc.).

— Unaligned loads and stores, avoiding asm.js's need to mask (which `asm.js` did for Typed Array compatibility purposes). This helps with practically every load and store.

— Various CPU instructions like popcount, copysign, etc. Each of these can help in specific circumstances (e.g. popcount can help in cryptanalysis).

— How much a specific benchmark benefits will depend on whether it uses the features mentioned above. We often see a 5 percent speedup on average compared to `asm.js`. Further speedups are expected in the future from CPU features like SIMD.

3. Toolchain Improvements.

WebAssembly is primarily a compiler target, and therefore has two parts: Compilers that generate it (the toolchain side), and VMs that run it (the browser side). Good performance depends on both.

This was already the case with `asm.js`, and Emscripten did a bunch of toolchain optimizations, running LLVM's optimizer and also Emscripten's asm.js optimizer. For WebAssembly, we built on top of that, but have also added some significant improvements while doing so. Both asm.js and WebAssembly are not typical compiler targets, and in similar ways, so lessons learned during the asm.js days helped do things better for WebAssembly. In particular:

— Mozilla replaced the Emscripten `asm.js` optimizer with the Binaryen WebAssembly optimizer, which is designed for speed. That speed lets us run more costly optimization passes. For example, they removed duplicate functions by default when optimizing, which often shrinks large compiled C++ codebases by around 5 percent.

— Better optimizations for irreducible and convoluted control flow, improving the Relooper algorithm.

— Overall, these toolchain improvements help about as much as moving from `asm.js` to WebAssembly.

4. Predictably Good Performance.

`asm.js` could run at basically native speed, but it never actually did so in all browsers consistently. The reason is that some tried to optimize it one way, some another, with differing results. Over time things started to converge, but the basic problem was that asm.js was not an actual standard: It was an informal spec of a subset of JavaScript, written by one vendor, that only gradually saw interest and adoption from the others.

CHAPTER 2
WEBASSEMBLY COMPILERS AND ENGINES OVERVIEW

As any other programming language ecosystem, WebAssembly has source to WAT or source to WASM compilers and IR interpreters (or runtimes). The most used source languages for WebAssembly are C/C++ and Rust, which are used for development of high-performant applications. Also JavaScript and TypeScript are popular for compiling to WebAssembly IR or binary code. This chapter contains short overview of the most efficient tools for WebAssembly compilation and execution.

2.1. Compilers

2.1.1. emscripten.

`emscripten` is the most popular and the most efficient LLVM to Web ahead-of-time compiler and debugger tools. It supports all languages which could be compiled to LLVM bitcode.

Originally, it had only JavaScript as a compilation target language, but in version `1.39`, `emscripten` develpers shipped new `binaryen` compiler infrastructure as part of `emscripten` tool. It allows fast, effective and easy compilation :

— `binaryen` has a single C header, where whole `C API` is described. This header could be used from JavaScript.

— `binaryen` designed as multicore tools for completely parallen code generation and optimization techniques.

— as a rule, `binaryen` compiler make a lot of compilation passes for better optimization. It uses both common and specific compilers optimization techniques as constant folding, loop fusion, dead code elimination, experimental escape analysis and target code minification.

— `binaryen` also includes WebAssembly interpreter and separate WebAssmbly IR optimizer.

Nowadays (year 2019) `emscripten` compiler infrastructure is the most popular among C/C++/Swift developers.

2.1.2. Rust WebAssembly target.

Rust compiler infrastructure supports Rust to WAT and Rust to WASM since version `1.30`. WAT and WASM are one of the backend output options. You can add the target by executing `rustup target add wasm32-unknown-unknown` in your terminal. Compilation of Rust program is possible by executing `cargo build -target wasm32-unknown-unknown -release`.

Rust optimized compiler has big featureset and it is one of the most efficient compiler nowadays.

2.1.3. Go WebAssembly target.

Go compiler infrastructure supports Go to WASM experimental compilation since version `1.11`.

2.1.4. AssemblyScript.

AssemblyScript is TypeScript-to-WASM compiler was written in TypeScript. It has its own compiler frontend and uses `binaryen` as a compiler backend. AssemblyScript team has a lot of demos and tutorials how to start working with AssemblyScript. Author of this thesis was used AssemblyScript compiler for compilation his own runtime test and benchmarks.

2.1.5. TeaVM. TeaVM is an ahead-of-time compiler for Java bytecode that emits JavaScript or WebAssembly that runs in a browser. Its close relative is the well-known GWT. The biggest difference is that TeaVM can use compiled Java class files. Moreover, the source code is not required to be Java, so TeaVM successfully compiles Kotlin and Scala. [10] Also, TeaVM produces fast, small JavaScript code for web apps that start quickly, even on mobile devices.

TeaVM tries to reconstruct original structure of a method, so in most cases it produces JavaScript that developers would write manually. TeaVM has a very sophisticated optimizer, which knows a lot about the code. Some examples are:

— Dead code elimination produces very small JavaScript.

— Devirtualization turns virtual calls into static function calls, which makes code faster.

— TeaVM can reuse one local variables to store several local variables.

— TeaVM renames methods to as short forms as possible; UglifyJS usually can't perform such optimization.

— TeaVM supports threads. TeaVM is capable of transforming methods to continuation-passing style. This makes possible to emulate multiple logical threads in one physical thread. TeaVM threads are, in fact, green threads.

2.2. Runtimes

The second part of WebAssembly ecosystem is runtimes. They are presented by browsers execution environments and WebAssembly VM for embedded and research.

2.2.1. Reference Interpreter.

WebAssembly specification contains a link to so-called *reference interpreter*. This is a tool which both W3C Working Group and Community Group are using for development of a new features prototypes. The reference interpreter is located by link [11]. Author of this thesis was used this interpreter to compare his implementation with the specification.

2.2.2. JavaScript virtual machines.

This section contains description of virtual machines which support WebAssembly binaries execution. Following the statistics, they are the platform with the highest usage of WebAssembly context.

2.2.2.1. V8.

Chrome V8, or V8, is a JavaScript engine developed by dutch Google office for Google Chrome and Chromium browsers. It is written on C++. The first version of the V8 engine was released September 2, 2008, the same date when first version of Google Chrome was released. It has also been used as query execution context in different NoSQL databases, in Node.js, which is JS exection plarform on the server-side and in Electron framework.

V8 has Ignition interpreter, which executes sources after VM initialization and collects heuristics for TurboFan JIT-compiler, which compiles JavaScript directly to native machine code, based on collected statistics. The compiled code is optimized dynamically at runtime, based on heuristics of the code's execution profile. Optimization techniques used include inlining, polymorphic inline caching and elision of expensive runtime properties; it also provides escape analysis and scalar replacement based on sea-of-nodes IR techniques. V8 contains a generational incremental collector and new experimental parallel and gerenational garbage collector called Orinoco. This engine uses Torque language for new features implementation within.

V8 can compile sources to x86 PowerPC, ARM and MIPS instructions set. [12]

Google Chrome supports WebAssembly 1.0 since version 57 (V8 version 5.7). Nowadays V8 has the most complete WebAssembly featureset including implementation level proposals.

2.2.2.2. SpiderMonkey.

SpiderMonkey is a JavaScript engine developed by Brendan Eich in Netscape Communications as first JavaScript execution engine. SpiderMonkey provides JavaScript support for Mozilla Firefox and various embeddings such as the GNOME 3 desktop, Yahoo Widgets or Adobe Acrobat.

SpiderMonkey has one interpreter which executes sources and collects heuristics during VM startupand two JIT-compilers : baseline and IonMonkey. They

have different optimization levels and they are designed for better balance between compilation time (IonMonkey produces highly optimized code, but spends more time than baseline compiler) and exectuion time. Baseline compiler uses optimizations like polymorphic inline caches, inlining, proxies, minor gc calls and common optimization techniques like constant folding etc.

SpiderMonkey has a mark-sweep-compact generational incremental garbage collection. Much of the collector work is performed on helper threads.

SpiderMonkey engine supports IA-32, x86-64, ARM, MIPS, SPARC architectures.

Mozilla Firefox supports WebAssembly 1.0 since version 52. SpiderMonkey implements all specifications features and some high-level proposals.

2.2.2.3. JSCore.

JavaScriptCore (simply JSCore) is a JavaScript engine developed in Apple. JavaScriptCore provides JavaScript support for Safari browser and various browsers based on WebKit engine and Steam (game delivery service from Valve).

JavaScriptCore has one interpreter which executes sources and collects heuristics during VM startup and three JIT-compilers : baseline, DFG and FTL. As SpiderMonkey, they have different optimization levels and they are designed for better balance between compilation time and exectuion time. Some experts think that JavaScriptCore has the most successful and promising architecture between all 4 major browsers.

JavaScriptCore contains a mark-compact generational garbage collection. Much of the collector work is performed on helper threads.

JavaScriptCore engine supports x86-64 and ARM architectures.

Safari supports WebAssembly 1.0 since version 11. Safari implements all specifications features and some high-level proposals.

2.2.2.4. ChakraCore.

ChakraCore is a JavaScript engine developed in Microsoft. JavaScriptCore provided JavaScript support for Internet Explorer, Microsoft Edge browsers,

Node.js-chakra and different databases engine. It is popular for embedded system due to extended support.

ChakraCore has one interpreter which executes sources and collects heuristics during VM startup and two JIT-compilers : SimpleJIT and FullJIT. They have different optimization levels and they are designed for better balance between compilation time and exectuion time. ChakraCore contains a mark-sweep generational garbage collection called `Recycler`. Much of the collector work is performed on helper threads. Also it has very interesting allocations algorithms called `ArenaAllocator`.

JavaScriptCore engine supports i386, x86-64 and ARM architectures.

Edge supports WebAssembly 1.0 since version 16, in return Internet Explorer doesn't suport WebAssembly. ChakraCore support only specifications features and a few important proposals, like atomics and threading. In February, 2019 Mirosoft announces that they would build next Edge browser versions based on Chrominum, but they will continue support of ChakraCore due to big popularity in embedded industry.

Author uses ChakraCore for creation of prototype of bulk operations implementation.

2.2.3. Other runtimes. There are also a couple of other runtimes and virtual machines which can compile and/or execute WebAssembly binaries. As a rule, such WebAssembly runtimes which pedantically follows the WebAssembly specification. They was created by enthusiasts who want to train them hard skills or who cares about WebAssemly as technology.

CHAPTER 3
RELATED WORK

3.1. Requirements for bulk operations

WebAssembly virtual machine should pedantically follows the WebAssembly specification for implementation of bulk memory operations. The chapters where the WebAssembly engine implementation must match the specification are :

— 2.5.4 Tables
— 2.5.5 Memories
— 3.3.4 Memory instructions
— 3.4.2 Tables
— 3.4.3 Memories
— 4.2.7 Table Instances
— 4.2.8 Memory Instances
— Table Section and Memory Section binary format.

3.2. Conditional segment initialization

n the MVP, segments are initialized during module instantiation. If any segment would be initialized out-of-bounds, then the memory or table instance is not modified.

This behavior is changed in the bulk memory proposal. Each active segment is initialized in module-definition order. For each segment, each byte in the data segment is copied into the memory, in order of lowest to highest addresses. If, for a given byte, the copy is out-of-bounds, instantiation fails and no further bytes in this segment nor further segments are copied. Bytes written before this point stay written.

The behavior of element segment initialization is changed similarly, with the

difference that elements are copied from element segments into tables, instead of bytes being copied from data segments into memories.

3.3. Memory store operations

[3]

Name	Opcode	Description
memory.init	`0xfc 0x08`	copy from a passive data segment to linear memory
memory.copy	`0xfc 0x0a`	copy from one region of memory to another region
memory.fill	`0xfc 0x0b`	fill a region of linear memory with a given byte value
table.init	`0xfc 0x0c`	copy from a passive element segment to a table
table.copy	`0xfc 0x0e`	copy from a passive data segment to linear memory
data.drop	`0xfc 0x09`	prevent further use of a passive element data segment
elem.drop	`0xfc 0x0d`	prevent further use of passive element segment

Table 3.1

Set of new memory and table store operations

3.3.1. `memory.init`.

The `memory.init` instruction copies data from a given passive segment into a target memory. The target memory and source segment are given as immediates.

The instruction has the signature `[i32 i32 i32] -> []`. The parameters are, in order:

— **first parameter :** destination address;

— **second parameter :** offset into the source segment

— **third parameter :** size of memory region in bytes; It is a validation error to use `memory.init` with an out-of-bounds segment index.

A trap occurs if:

— the segment is used after it has been dropped via data.drop. This includes active segments that were dropped after being copied into memory during module instantiation;

— any of the accessed bytes lies outside the source data segment or the target memory;

— the source offset is greater than the length of the source data segment;

— the destination offset is greater than the length of the target memory;

Note: it is allowed to use `memory.init` on the same data segment more than once.

Initialization takes place bytewise from lower addresses toward higher addresses. A trap resulting from an access outside the source data segment or target memory only occurs once the first byte that is outside the source or target is reached. Bytes written before the trap stay written.

Data are read and written as-if individual bytes were read and written, but various optimizations are possible that avoid reading and writing only individual bytes.

Note: the semantics require bytewise accesses, so a trap that might result from, say, reading a sequence of several words before writing any, will have to be handled carefully: the reads that succeeded will have to be written, if possible.

3.3.2. `memory.fill`.

Set all bytes in a memory region to a given byte. This instruction has an immediate argument of which memory to operate on, and it must be zero for now.

The instruction has the signature `[i32 i32 i32] -> []`. The parameters are, in order:

— **first parameter :** destination address;

— **second parameter :** byte value to set

— **third parameter :** size of memory region in bytes;

A trap occurs if:

— any of the accessed bytes lies outside the target memory;

— the destination offset is greater than the length of the target memory;

Filling takes place bytewise from lower addresses toward higher addresses. A

trap resulting from an access outside the target memory only occurs once the first byte that is outside the target is reached. Bytes written before the trap stay written.

Data are written as-if individual bytes were written, but various optimizations are possible that avoid writing only individual bytes.

3.3.3. `memory.copy`.

Copy data from a source memory region to destination region. he regions are said to overlap if they are in the same memory and the start address of one region is one of the addresses that's read or written (by the copy operation) in the other region.

This instruction has two immediate arguments: the source and destination memory indices. They currently both must be zero. If the regions overlap, and the source region starts at a lower address than the target region, then the copy takes place as if from higher to lower addresses: the highest source address is read first and the value is written to the highest target address, then the next highest, and so on. Otherwise, the copy takes place as if from lower to higher addresses: the lowest source address is read first and the value is written to the lowest target address, then the next lowest, and so on. (The direction of the copy is defined in order to future-proof memory.copy for shared memory and a memory read/write protection feature.)

The instruction has the signature `[i32 i32 i32] -> []`. The parameters are, in order:

— **first parameter :** destination address;

— **second parameter :** source address

— **third parameter :** size of memory region in bytes;

A Trap occurs if :

— any of the accessed bytes lies outside the source data segment or the target memory;

— the source offset is greater than the length of the source data segment;

— the destination offset is greater than the length of the target memory;

A trap resulting from an access outside the source or target region only occurs once the first byte that is outside the source or target is reached (in the defined copy order). Bytes written before the trap stay written.

Data are read and written as-if individual bytes were read and written, but various optimizations are possible that avoid reading and writing only individual bytes.

3.4. Table store operations

3.4.1. `table.fill`.

Set all bytes in a table region to a given byte. This instruction has an immediate argument of which table to operate on, and it must be zero for now.

The instruction has the signature `[i32 i32 i32] -> []`. The parameters are, in order:

— **first parameter :** destination address;

— **second parameter :** byte value to set

— **third parameter :** size of memory region in bytes;

A trap occurs if:

— any of the accessed bytes lies outside the target table;

— the destination offset is greater than the length of the target table;

3.4.2. `table.copy`.

Copy data from a source table region to destination region. he regions are said to overlap if they are in the same table and the start address of one region is one of the addresses that's read or written (by the copy operation) in the other region.

This instruction has two immediate arguments: the source and destination table indices. They currently both must be zero. If the regions overlap, and the source region starts at a lower address than the target region, then the copy takes place as if from higher to lower addresses: the highest source address is read first

and the value is written to the highest target address, then the next highest, and so on. Otherwise, the copy takes place as if from lower to higher addresses: the lowest source address is read first and the value is written to the lowest target address, then the next lowest, and so on.

The instruction has the signature `[i32 i32 i32] -> []`. The parameters are, in order:

— **first parameter :** destination address;

— **second parameter :** source address

— **third parameter :** size of table region in bytes;

A Trap occurs if :

— any of the accessed bytes lies outside the source element segment or the target table;

— the source offset is greater than the length of the source table;

— the destination offset is greater than the length of the target table;

A trap resulting from an access outside the source or target region only occurs once the first byte that is outside the source or target is reached (in the defined copy order). Bytes written before the trap stay written.

3.5. Passive memory store prevention

3.5.1. `data.drop`.

The `data.drop` instruction prevents further use of a given segment. After a data segment has been dropped, it is no longer valid to use it in a `memory.init` instruction. This instruction is intended to be used as an optimization hint to the WebAssembly implementation. After a memory segment is dropped its data can no longer be retrieved, so the memory used by this segment may be freed. It is a validation error to use data.drop with an out-of-bounds segment index.

A trap occurs if the segment was already dropped. This includes active segments that were dropped after being copied into memory during module instantiation.

3.5.2. `table.drop`.

The `elem.drop` instruction prevents further use of a given segment. After a element segment has been dropped, it is no longer valid to use it in a `table.init` instruction. This instruction is intended to be used as an optimization hint to the WebAssembly implementation. After a element segment is dropped, the table used by this segment may be freed. It is a validation error to use `elem.drop` with an out-of-bounds segment index.

A trap occurs if the segment was already dropped. This includes active segments that were dropped after being copied into table during module instantiation.

3.6. SIMD bulk operations

There are a processors where SIMD operations available. WebAssembly has linear memory, which allows us to use SSE or AVX extensions. Author created an experimental solution where copying avoids caches and use all SIMD memory to provide fast memory storing. `movaps` instruction moves a double quadword containing four packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. Author used it for partial loading from source.

`movntps` instruction moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception will be generated. Author used it for flushing the

copied memory to the destination.

CHAPTER 4
EXPERIMENTAL TESTING

4.1. Hardware and Software setup

Author used two devices for his investigation :

1. Personal computer (PC) with Intel Core i5-7600 processor (Caby Lake architecure) with enabled hyperthreading and enabled SIMD support, 16 GB of HDDR4 HyperX RAM and 128 GB Kingston SSD. PC is operated by Windows 10 OS. ChakraCore was compiled with `mvsc 2017` compiler using both Debug and Release configurations.
2. Laptop MacBook Pro 2015 with Intel Core i5-4*** processor of Haswell series with enabled hyperthreading and SIMD support. MacBook has 8 GB of DDR3 RAM and 128 GB Apple SSD. It operates by MacOS 10.14.2. ChakraCore was compiled with clang 9.0 using test, debug and release builds.

4.2. Compile time results

Compile time result will be compared between 5 different builds on two different platforms : Mac and Windows.

Table 4.1

Compile-time measurement result

Build:	Debug	Release	Test
MacOS:	28 mins	22 mins	-
PC	7 mins	5.5 mins	6.5 mins

4.3. Run time tests and benchmarks

Let's measure throughput of implemented algorithms :

1. First benchmark is measurement and comparison of throughput of our `memory.copy` operation using MacBook. It is copying one gigabyte of some memory filler with different ranges of data size and using different copy techniques. Data size varies from 32 bytes to 1 megabyte. Count of elements varies from 1024 (one chuck has 1 Mb) to 33 millions (chunk size is 32 bytes). Before the main measurement benchmark also has warmup phase, where 100 megabytes of memory copies to other destination. The result is presented in table below.

Size, count	memory.copy	i64 x4	i64 x2	i32 x4	i32 x2
32b, N=33554432	1.812	1.036	1.021	0.783	1.097
64b, N=16777216	3.459	1.670	1.580	1.068	1.733
128b, N=8388608	6.832	2.387	2.175	1.319	2.388
256b, N=4194304	13.724	3.013	2.675	1.434	2.731
512b, N=2097152	27.337	3.460	2.781	1.566	3.150
1.0Kb,N=1048576	55.630	2.888	2.532	1.277	2.802
2.0Kib, N=524288	111.520	3.798	3.191	1.390	3.629
4.0Kib, N=262144	223.065	3.858	3.343	1.334	3.606
8.0Kib, N=131072	445.831	3.966	3.280	1.485	3.697
16.0Kib, N=65536	484.956	4.006	2.806	1.658	2.705
32.0Kib, N=32768	602.564	4.106	3.408	1.696	3.131
64.0Kib, N=16384	795.975	2.958	2.711	1.677	3.719
128.0Kib, N=8192	735.602	4.038	3.412	1.711	3.806
256.0Kib, N=4096	733.333	4.105	3.423	1.665	3.166
512.0Kib, N=2048	277.778	3.078	2.927	1.272	2.842
1.0Mib, N=1024	344 .759	3.253	3.181	1.557	3.504

2. Second benchmark is measurement and comparison of throughput of our `memory.fill` operation using MacBook. It is storing one gigabyte of random values with different ranges of data size. Data size varies from 32 bytes to 256 kbytes. 256 Kb) to 33 millions (chunk size is 32 bytes). Before the main measurement benchmark also has warmup phase, where 100 megabytes of memory stores with random numeric value. The result is presented in table below.

№	Size, count	memory.fill	i64	f64
1	32b, N=33554432	437.541	231.047	202.857
2	64b, N=16777216	444.747	219.208	195.645
3	128b, N=8388608	409.110	236.186	200.928
4	256b, N=4194304	403.934	255.719	233.854
5	512b, N=2097152	416.871	235.361	209.014
6	1.0Kib, N=1048576	425.176	215.470	198.963
7	2.0Kib, N=524288	428.753	220.915	181.198
8	4.0Kib, N=262144	407.274	210.355	180.818
9	8.0Kib, N=131072	409.703	199.489	172.027
10	16.0Kib, N=65536	408.377	194.998	171.126
11	32.0Kib, N=32768	400.511	200.759	147.658
12	64.0Kib, N=16384	402.202	168.391	147.151
13	128.0Kib, N=8192	397.917	190.240	154.286
14	256.0Kib, N=4096	413.127	274.415	161.437

Table 4.2

Measurement of `memory.fill` operations and its rivals

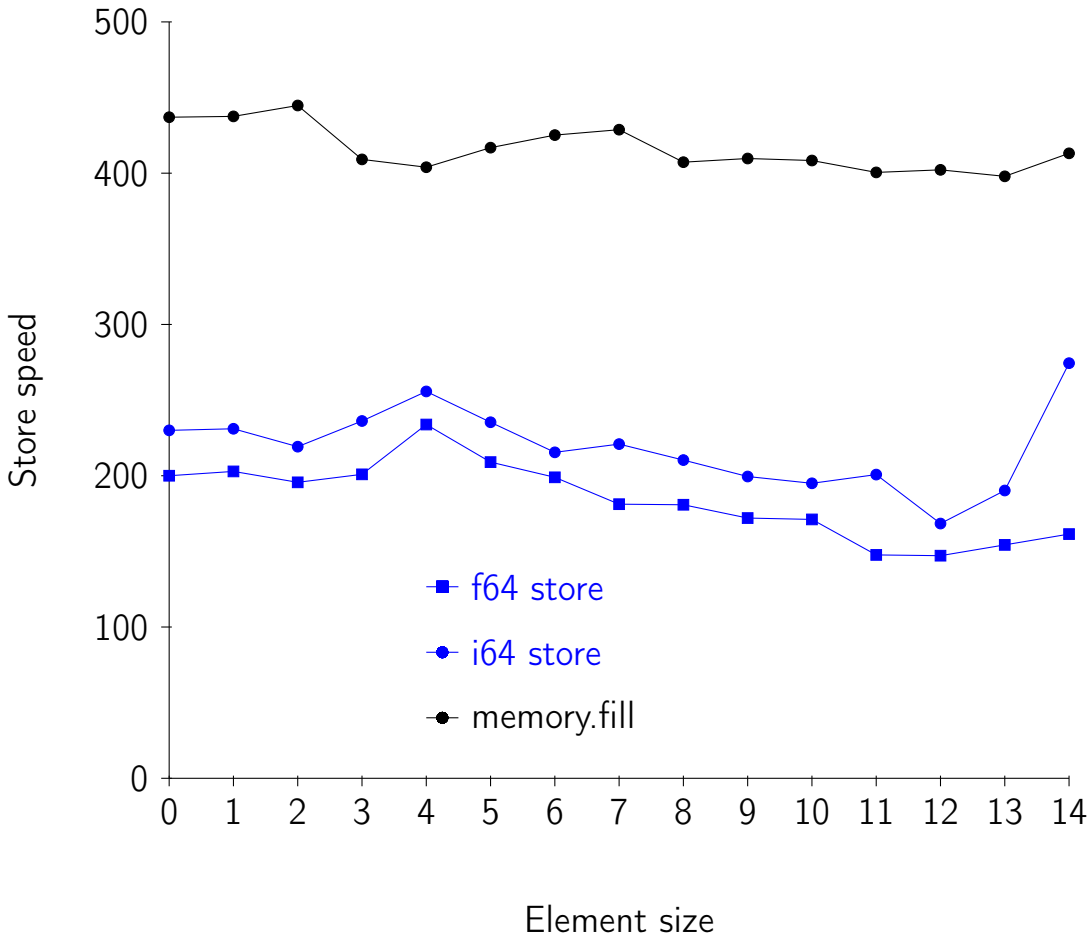

Figure 4.1: Plot with benchmark results of memory.fill

3. Third benchmark is measurement and comparison of throughput of our `memory.init` operation using MacBook. It is storing one gigabyte of random values with different ranges of data size during the passive segment initializing. Data size varies from 32 bytes to 256 kbytes. 256 Kb) to 33 millions (chunk size is 32 bytes). It has the same warmup setup as previous benchmark.

№	Size, count	memory.init	i64	f64
1	32b, N=33554432	453.3	217.6	212.2
2	64b, N=16777216	413.1	211.1	203.2
3	128b, N=8388608	417.8	214.4	206.6
4	256b, N=4194304	420.0	221.2	232.2
5	512b, N=2097152	476.7	231.5	199.3
6	1.0Kib, N=1048576	507.5	355.8	274.9
7	2.0Kib, N=524288	465.9	229.1	199.4
8	4.0Kib, N=262144	581.2	317.7	249.2
9	8.0Kib, N=131072	437.0	217.6	210.6
10	16.0Kib, N=65536	413.3	249.9	286.3
11	32.0Kib, N=32768	404.0	230.7	257.7
12	64.0Kib, N=16384	411.2	212.8	233.0
13	128.0Kib, N=8192	408.6	214.7	207.8
14	512.0Kib, N=2048	282.7	163.9	155.1

Table 4.3

Measurement of `memory.init` operations and its rivals

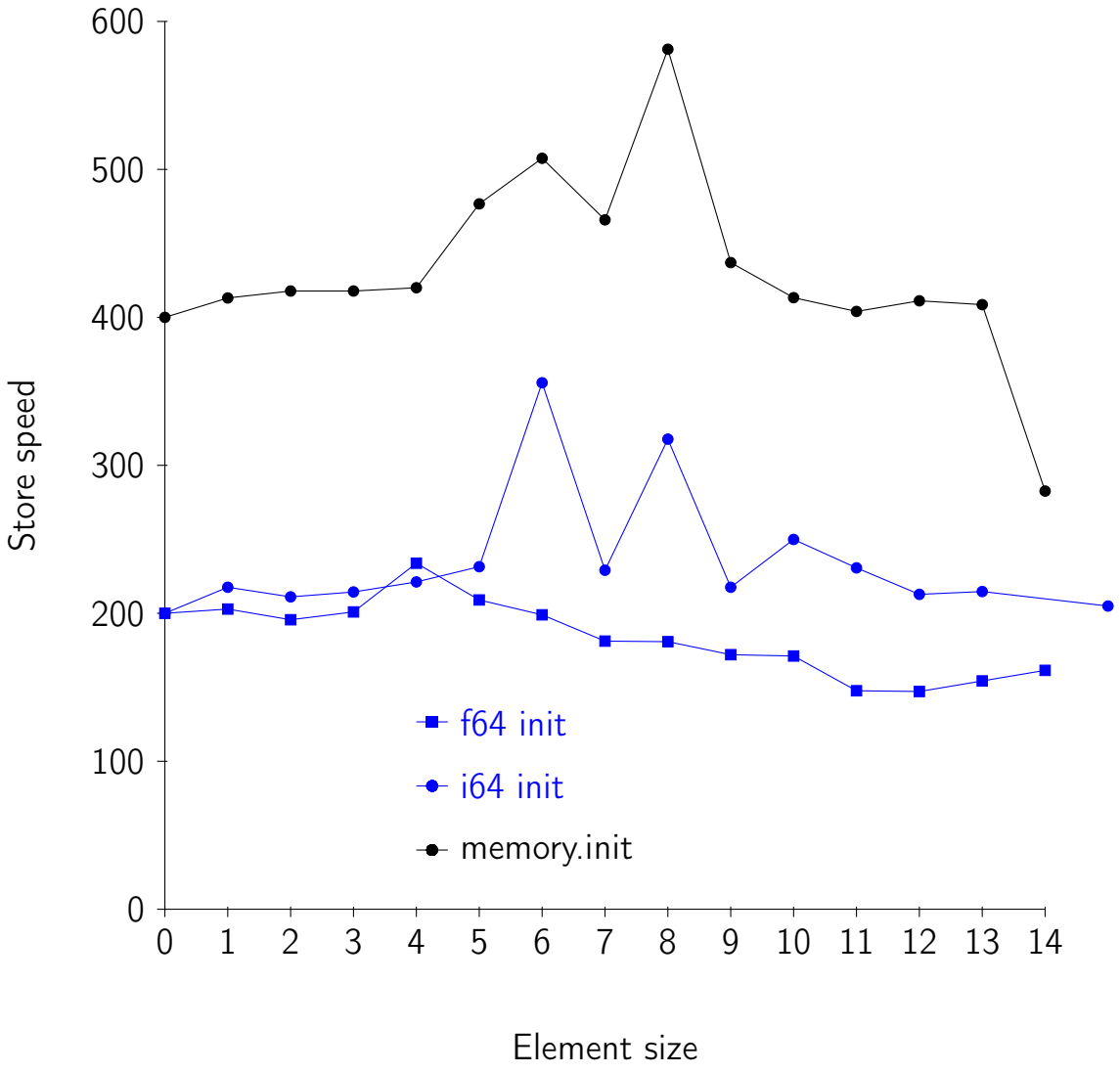

Figure 4.2: Plot with benchmark results of memory.init

4.4. Conclusions

While there has been extencive work done on measuring bulk memory operations for a variety of scenarios and implementations. An analysis of the time spent and operations executred revealed showed that the implemented algorithms of accelerating copying, initializing and filling of linear memory have given a great boost. For example, throughput of copying operation sugnificantly increased in 100...400 times! Other implemented bulk operations also haven given a boost in 2...3 times. These results are reported on all algorithms prototypes can be successfully applied to provided virtual machine and they are ready for implementation production mode.

CONCLUSIONS

In this thesis, we addressed the problem of slow bulk memory copying and filling in different WebAssembly runtimes. One of the main contributions of our work is design and prototyping of bulk memory operations ChakraCore engine, which supports WebAssembly. A discussion on different representations of individuals has been provided. In particular, we proposed additional optimization layer for copying and filling operations using SIMD featureset. Some of the constraints imposed to the matching could be introduced directly in the representations. Different types of bulk memory operations implememtation have been presented. Our contribution here is twofold. First an experimental comparison of their behavior has been performed, and second new optimizations have been designed. Second, passive segment initialization was designed and developed in concrete runtime. The main focus of our thesis was on the optimization using SIMD registers to avoid caching. This contribution allows now to use bulk operations in compilers for straight mapping of compiled source code. From an experimental point of view, our contribution lies in the comparison of the performance of bulk operations in both SIMD and non-SIMD strategies. These differences in the results have been proved to be statistically significant after measurement tests.

BIBLIOGRAPHY

[1] Wikipedia, "Webassembly," May 2019.

[2] *WebAssembly Specification*, 1st ed., WebAssembly Working Group, 4 2019, https://webassembly.github.io/spec/core/bikeshed/index.html.

[3] *WebAssembly Working Group*, 1st ed., W3C, 8 2017, https://www.w3.org/2017/08/wasm-charter.

[4] *WebAssembly Community Group*, 37th ed., W3C, 5 2019, https://webassembly.github.io/cg-charter/.

[5] Wikipedia, "Java applet," 2019, wikipedia page.

[6] ——, "Microsoft silverlight," 2019, wikipedia page.

[7] ——, "Adobe flash," 2019, wikipedia page.

[8] *What is asm.js, exactly?*, MDN, 7 2018, the Mozilla Developer Network article.

[9] A. Zakai, *Why WebAssembly is faster than asm.js*, 1st ed., Mozilla Corp and Mozilla Foundation, 3 2017, https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/.

[10] *Reference interpreter*, 1st ed., W3C, 5 2019, webAssembly's reference interpreter sources and papers.

[11] *TeaVM*, 1st ed., TeaVM, 5 2019, webAssembly's compiler.

[12] *Google V8*, 1st ed., Wikipedia, 5 2019, google V8 Engine.

Appendix A

Example of compiled S-expression

Compilation result (C++ to wat) of example from introduction.

```
(module
 (type $FUNCSIG$ii (func (param i32) (result i32)))
 (type $FUNCSIG$iiii (func (param i32 i32 i32) (result i32)))
 (import "env" "_Znaj" (func $_Znaj (param i32) (result i32)))
 (import "env" "memcpy" (func $memcpy (param i32 i32 i32) (result i32)))
 (import "env" "memset" (func $memset (param i32 i32 i32) (result i32)))
 (table 0 anyfunc)
 (memory $0 1)
 (export "memory" (memory $0))
 (export "_Z12testFunctionPi" (func $_Z12testFunctionPi))
 (func $_Z12testFunctionPi (; 3 ;) (param $0 i32) (result i32)
  (local $1 i32) (local $2 i32) (local $3 i32) (local $4 i32)
  (set_local $1
   (call $memset
    (call $_Znaj
     (i32.const 4096)
    )
    (i32.const 10)
    (i32.const 1024)
   )
  )
  (set_local $2
   (call $_Znaj
    (i32.const 4096)
   )
  )
  (i32.store
   (get_local $1)
   (i32.const 168430091)
```

```
  )
  (set_local $4 (i32.const 4))
  (loop $label$0
   (i32.store
    (tee_local $3
     (i32.add
      (get_local $1) (get_local $4)
     )
    )
    (i32.add
     (i32.load
      (get_local $3)
     )
     (i32.const 1)
    )
   )
   (br_if $label$0
    (i32.ne
     (tee_local $4
      (i32.add
       (get_local $4)
       (i32.const 4)
      )
     )
     (i32.const 4096)
    ))
  )
  (i32.load
   (call $memcpy
    (get_local $2) (get_local $1)
    (i32.const 1024)
   )
  )
 )
)
```

Appendix Б

Trademarks

The follwing terms are trademarks or registered trademarks of Alphabet Inc. in the United States or other countries : Google, Google Chrome, V8. PowerPC is a registered trademark of International Business Machines Corporation in the United States or other countries. Node.js is a registered trademark of Node.js Foundation in the United States or other countries. Wikipeida is a registered trademark of Wikipeida Foundation in the United States or other countries.