SABLEWASM: A STATIC COMPILER AND RUNTIME FOR WEBASSEMBLY

Hongji Chen, School of Computer Science

McGill University, Montreal

August, 2021

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Computer Science

©Hongji Chen, 2021

Abstract

WebAssembly is a relatively new language, introduced to improve the performance of compute-intensive workloads in web-based applications. It offers a compact binary byte-code intended to allow for fast compilation and improved optimization opportunities over dynamic web languages like JavaScript. These properties, however, also make it an interesting target for static execution, enabling web code to run outside of a browser as well as within it. In this thesis, we describe *SableWasm*, a static, multi-pass compiler system that translates sandboxed WebAssembly applications to native shared libraries. Our work covers several different aspects of compiler design. First, we provide an efficient and extensible WebAssembly module parsing and validation framework, with improved execution speed and memory footprint compared to the reference baseline. We then define a middle-level intermediate representation and build an analysis and transformation framework. We explore several classic data-flow analyses, such as dominator-tree construction within the framework, and additionally identify several WebAssembly specific optimization opportunities, which we address through custom transformation passes. SableWasm also incorporates several in-progress extension proposals including the SIMD vector operation extension. Optimized intermediate code is then converted to native code through a backend implementation with the help of the LLVM compiler framework and a runtime that enables C/C++ programs to interact with the WebAssembly module directly. Finally, we evaluate SableWasm by benchmarking against several well-known testing suites and observe performance improvement compared to the baseline implementation.

Abrégé

WebAssembly est un langage relativement nouveau, introduit pour améliorer les performances des charges de travail gourmandes en calcul dans les applications Web. Il offre un bytecode binaire compact destiné à permettre une compilation rapide et des opportunités d'optimisation améliorées par rapport aux langages Web dynamiques comme JavaScript. Ces propriétés, cependant, en font également une cible intéressante pour l'exécution statique, permettant au code Web de s'exécuter à l'extérieur d'un navigateur ainsi qu'à l'intérieur de celui-ci. Dans cette thèse, nous décrivons *SableWasm*, un système de compilateur statique multi-passes qui traduit les applications WebAssembly en bac à sable en bibliothèques partagées natives. Notre travail couvre plusieurs aspects différents de la conception d'un compilateur. Premièrement, nous fournissons un cadre d'analyse et de validation de module WebAssembly efficace et extensible, avec une vitesse d'exécution et une empreinte mémoire améliorées par rapport à la ligne de base de référence. Nous définissons ensuite une représentation intermédiaire de niveau intermédiaire et construisons un cadre d'analyse et de transformation. Nous explorons plusieurs analyses de flux de données classiques, telles que la construction d'arbres dominants dans le cadre, et identifions en outre plusieurs opportunités d'optimisation spécifiques à WebAssembly, que nous abordons via des passes de transformation personnalisées. SableWasm intègre également plusieurs propositions d'extension en cours, y compris l'extension d'opération vectorielle SIMD. Le code intermédiaire optimisé est ensuite converti en code natif via une implémentation backend à l'aide du framework de compilateur LLVM et d'un runtime qui permet aux programmes C/C++ d'interagir directement avec le module WebAssembly. Enfin, nous évaluons SableWasm en comparant plusieurs suites de tests bien connues et observons une amélioration des performances par rapport à la mise en œuvre de base.

Acknowledgements

First, I would like to extend my deepest gratitude to Professor Clark Verbrugge. The work would not have been possible without his support and advice, especially during a global pandemic. Second, I would like to thank Professor Laurie Hendren for her guidance in the field of compiler design in my early days as an undergraduate student. Finally, I would like to thank my colleagues and friends, especially my Sable lab mates, for their encouragement throughout the entire thesis journey.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Web-based applications have grown in popularity in recent years. From the early days of simple web applets to current full-blown programs, their codebase's complexity and size has grown rapidly. Due to the design of most browsers, programmers have to choose JavaScript or its dialects to implement them. This approach is quite successful; however, it still leaves several problems unsolved. First, JavaScript is a scripting language and employs many dynamic features that prevent backend runtime environment from efficient execution, such as dynamic typing. Additionally, when porting existing applications to JavaScript, especially those with a large codebase where manually translating source code line-by-line is not feasible, a nontrivial source-to-source compiler is needed due to the structural difference between native binaries and JavaScript source codes.

To address these problems, the WebAssembly working group was established in 2017 and proposed a new standard for distributing applications over the Internet. WebAssembly focuses on safety, performance, portability and module compactness. These properties also make it an interesting target for static execution, enabling sandboxed applications outside of browsers. Although WebAssembly started as an intermediate format for distributing compiled programs over the Internet, it is not limited to the Web. The WebAssembly working group, in its famous WebAssembly introduction paper [8], also shows that WebAssembly can serve as an open standard for embedding in a multi-

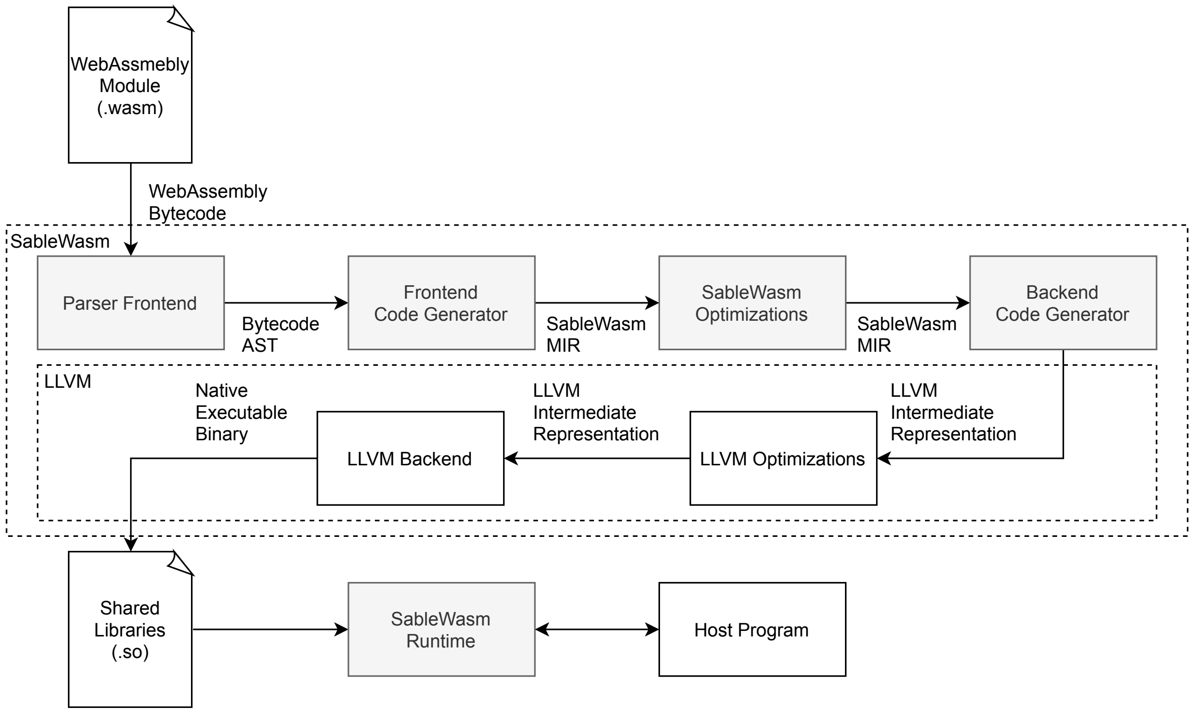


Figure 1.1: The SableWasm compiler and runtime

contexts environment. To this end, the WebAssembly community further designed the WebAssembly System Interface (WASI), which provides a standardized interface for WebAssembly modules to access native features such as the file system.

WebAssembly is also an evolving language. Although the WebAssembly community has published the minimum viable product (MVP) WebAssembly, the community is still actively proposing and experimenting with new language features, such as exception handling and garbage collection. These additional language features are proposed in language extension proposals that modify the current WebAssembly specification syntactically and semantically. Thus, a well-designed WebAssembly runtime environment system should be modular and extensible, leaving space for future design changes.

1.1 Contribution

This thesis aims to design and implement a runtime environment that enables WebAssembly to run outside of the browser. To this end, this thesis makes three major contributions.

Figure 1.1 illustrates the SableWasm compiler and runtime system. We mark our contributions in this thesis as shaded boxes in the figure.

Implementing a WebAssembly runtime system Our first contribution is a standalone WebAssembly runtime environment with support for the WebAssembly System Interface (WASI). We first start by implementing a custom extensible parser frontend for WebAssembly binary format, shown as the 'Parser Frontend' in figure 1.1. We then define a 'middle-level' representation (MIR) for SableWasm. To match modern hardware, SableWasm MIR is a register-based *control flow graph* representation of the program, while, on the other hand, WebAssembly operates over a stack-based virtual machine. Hence, translating between them is nontrivial. Therefore, we design and implement a frontend code generator that lowers WebAssembly bytecode into SableWasm MIR, shown as the 'Frontend Code Generator' in figure 1.1. SableWasm MIR plays a critical role in the SableWasm system. First, it provides a middle ground where we implement an extensible and straightforward optimization framework. With the help of the framework, we experiment with several analyses and optimizations on SableWasm MIR. Second, SableWasm MIR also separates the frontend from the backend. Currently, we implement an ahead-of-time (AOT) compiler backend using the LLVM compiler infrastructure [15], shown as the 'Backend Code Generator' in figure 1.1. However, there are several challenges when lowering SableWasm MIR into LLVM intermediate representation. For example, SableWasm MIR, similar to WebAssembly bytecode, utilizes several abstract high-level concepts such as linear memory and indirect function calls. These operations cannot be trivially mapped to LLVM instructions and require runtime library support. Hence, the last component of SableWasm is a runtime library that provides builtin runtime functions for the generated modules and defines an easy-to-use interface for the host system, shown as the 'SableWasm Runtime' in figure 1.1.

Adding support for WebAssembly extensions Our second contribution in this thesis is to experiment and adopt several in-progress WebAssembly language extensions. Sable-

Wasm is designed to be extensible and currently implements four post-MVP WebAssembly features. The most interesting one among them is perhaps the fixed-width SIMD operation extension which defines vector-based operations that can operate on multiple data simultaneously, packed into special vector registers and supported by modern hardware. The SIMD extension in WebAssembly introduces one additional value type and approximately 240 new instructions to the specification. As we have discussed earlier in this section, SableWasm MIR provides a middle ground where we perform optimization on the program. Therefore, we would like to keep the size of the SableWasm MIR instruction set simple. To achieve this goal, we carefully design a set of reduction patterns in the frontend code generator that significantly reduce the number of instructions needed. We also generalize our backend code generator that targets LLVM by emitting corresponding vector operation instructions.

Evaluating system performance Our last contribution in this thesis is to investigate how SableWasm performs and the factors that affect the performance. Here we focus on three research questions: First, how does SableWasm perform comparing to other existing WebAssembly runtime implementations? Second, does optimization over the input WebAssembly modules affect SableWasm's overall performance? Finally, does the SIMD operation extension bring performance improvement to the system? To answer these questions, we analyze the performance of three well-known benchmark suites, Polybench [36], Ostrich [9], and NPB [33]. We also examine generated LLVM intermediate representations in SableWasm to search for factors contributing to the slow down in the system.

1.2 Thesis outline

This thesis consists of nine chapters in total, including the introduction chapter. Chapter 2 discusses the background information that helps the understanding rest of the thesis. It first presents the motivation for WebAssembly and WebAssembly System Interface

(WASI), followed by a brief overview of the LLVM intermediate representation. Chapter 3 to chapter 6 discusses the design of implementation of the SableWasm system. Chapter 3 starts with presenting the custom extensible and efficient parser frontend for WebAssembly binary format. Chapter 4 continues the discussion of SableWasm by describing SableWasm MIR's design. Chapter 5 discusses the code generating strategies used when lowering WebAssembly bytecode to SableWasm MIR and the optimization framework. Chapter 5 also presents several optimization passes we experimented with the framework, such as control flow graph simplification and type inference. Chapter 6 illustrates the last component of SableWasm, the LLVM backend and the runtime support library. In chapter 7, we investigate the performance of SableWasm by presenting benchmark results and discussing several possible theories for the slowdown. Finally, chapter 8 discusses related work and chapter 9 presents our conclusion along with future work.

Chapter 2

Background

This chapter provides background information that helps to understand the thesis. We first revisit the rise of *asm.js* and its toolchain, *Emscripten*, followed by an introduction to *WebAssembly* and its standardized *WebAssembly System Interface* (WASI). Finally, we will give a brief overview of the LLVM compilation framework.

2.1 Emscripten and Asm.js

In the past decade, web-based applications are gaining popularity, and due to the design of most browsers, programmers tend to choose JavaScript or its dialects to implement them. One natural problem is how to compile programs that target the native platform to run over the internet. Making the situation more challenging, programs with a large codebase, such as games requiring complex video and physical computation, are nearly impossible to translate line-by-line manually. In 2010, Alon Zakai started the first attempt at translating source code that targets native platforms into JavaScript [34]. After two years of development, he published Emscripten that translates LLVM intermediate representation into asm.js, a JavaScript subset [37]. An asm.js program shares a similar programming model to that which one would expect on the native platform. The detailed asm.js specification is available on the official website [1]. We will visit several critical features in asm.js with examples in figure 2.1 (page 8). These examples are implementations of the Adler-32 hashing algorithm used in ZLib compression library [4] [2], in both C and its corresponding generated asm.js with Emscripten.

Function prologue and type annotation JavaScript is a dynamically typed language. Hence, a proper implementation needs to verify the types of variables when needed. Although several optimization techniques can eliminate some of the checks and improve the execution performance, such language features can still incur a significant performance loss. Asm.js adds type annotations to function parameters and expressions to address this problem. In figure 2.1b, Emscripten generates parameter annotations for parameters `$0_1` and `$0_2` at line 2 and line 3 respectively. The trailing bitwise 'or' operation against zero hints that both arguments are integral values since bitwise operations are only defined for integral values in JavaScript. Emscripten also annotates float-point numbers with the unary positive operation, '+', which we do not show in the example. A system that supports asm.js directly can quickly recover the type information from the annotations, which, in theory, can improve both the compilation and execution performance. On the other hand, for a system that does not recognize asm.js, the program above is still a valid JavaScript program, and the type annotations ensure the correct semantics for numeric operations.

Control flow LLVM employs a register-based intermediate representation with a *control flow graph* (CFG). However, JavaScript uses structured control flow and does not allow arbitrary jump statements similar to one would expect in C. Hence, when translating LLVM IR to asm.js, Emscripten mimics the branch instructions between basic blocks in the generated code with JavaScript control-flow statements. Emscripten uses a pattern-based translation and classifies control flow changes into three categories. In figure 2.1b,

[1]asm.js specification: `http://asmjs.org/spec/latest/`
[2]Revisiting Fletcher and Adler Checksums:
`http://www.zlib.net/maxino06_fletcher-adler.pdf`

```c
uint32_t adler32(void const *buffer_, size_t length_) {
  uint8_t const *buffer = (uint8_t const *)buffer_;
  uint32_t a = 1;
  uint32_t b = 0;
  for (size_t i = 0; i < length_; ++i) {
    a = (a + buffer[i]) % 65521;
    b = (a + b) % 65521;
  }
  return (b << 16) | a;
}
```

(a) C

```javascript
function $adler32($0_1, $1_1) {
  $0_1 = $0_1 | 0;
  $1_1 = $1_1 | 0;
  var $2_1 = 0, $3_1 = 0, $4_1 = 0;
  $3_1 = 1;
  if ($1_1) {
    while (1) {
      $3_1 = (HEAPU8[$0_1 + $2_1 | 0] + $3_1 >>> 0) % 65521 | 0;
      $4_1 = ($4_1 + $3_1 >>> 0) % 65521 | 0;
      $2_1 = $2_1 + 1 | 0;
      if (($2_1 | 0) != ($1_1 | 0)) { continue }
      break;
    };
    $2_1 = $4_1 << 16;
  }
  return $2_1 | $3_1;
}
```

(b) asm.js

```
(type $t0 (func (param i32 i32) (result i32)))
(func $adler32 (export "adler32") (type $t0)
    (local $l0 i32) (local $l1 i32)
    get_local $p1
    if $I0
      (set_local $l1 (i32.const 1))
      loop $L1
        (i32.rem_u
          (i32.add
            (get_local $l1)
            (i32.load8_u (get_local $p0)))
          (i32.const 65521))
        ... ...
      end
      (i32.or
        (i32.shl (get_local $l0) (i32.const 16))
        (get_local $l1))
      return
    end
    i32.const 1)
(memory $memory (export "memory") 2)
```

(c) Text-format WebAssembly

Figure 2.1: Adler 32 in C, asm.js and text-format WebAssembly

we demonstrate two of the three control-flow structures, *if* and *loop*, at line 6 and line 7 respectively. Asm.js also has a third control flow structure, *block*, which we do not show in the example. A *block* structure is similar to a *loop* structure and can be translated to a while loop with an always-false condition. A branch instruction referring to the *block* is equivalent to a break statement in this case. WebAssembly adopts a similar design, and we will revisit this in the later section with more details.

Byte array as heap Emscripten uses multiple typed array views that share a single underlying byte array buffer to simulate the heap in a native programming model. In figure 2.1b, the asm.js example uses `HEAPU8`, an unsigned byte view over the byte array at line 8, to access the data passed by the pointer via the first argument. Asm.js also offers other array views such as `HEAPI32` and `HEAPF32` which allows programs to access 32-bit signed integers and single-precision floating-point numbers on the heap. This technique also inspires the linear memory design in WebAssembly, which we will discuss later in the chapter with more details.

Emscripten is quite successful. Experiment results show that it can port most of the C/C++ programs of non-trivial code size to the web with approximately 50-67% of native performance [3] without any missing significant features.

2.2 WebAssembly

Although Emscripten with asm.js is successful, there are still several problems that remain unaddressed. One of them is the parsing overhead. As asm.js is a strict subset of JavaScript, parsing the generated program is a non-trivial task due to the complexity of JavaScript grammar. Additionally, because Emscripten emits generated programs in asm.js, the output size grows significantly faster than the native binary. Another problem regards the generated programs' safety, especially when running an untrusted module

[3] Alon Zakai's presentation on Emscripten at CppCon:
`https://kripken.github.io/mloc_emscripten_talk/cppcon.html`

received over the internet. In 2017, the WebAssembly community established and proposed a new standard for distributing programs over the internet to address these problems. The design of WebAssembly focuses on safety, performance, portability, and compactness. The introduction paper describes the detailed structure, validation rules [35], and execution semantics of WebAssembly [8]. Here we will only visit some of the key points that help understand the rest of the thesis. In figure 2.1c we also present a simple WebAssembly program that implements the Adler32 hashing.

Module structure WebAssembly modules can have four different kinds of entities: *functions*, *indirect tables*, *linear memories*, and *globals*. Modules are also able to import and export entities by names. In figure 2.1c, we define a *function* and a *linear memory* and export them under name `adler32` and `memory` respectively. WebAssembly functions can define an arbitrary number of local variables and a possibly empty sequence of instructions as the body. All instruction operates over an implicitly declared stack. The control-flow will return from the function by either a `return` instruction or reaching the end of the body. WebAssembly linear memories have bounds consisting of a pair of integers, representing the lower bound and upper bound respectively, [4] in units of 16-KiB pages. In figure 2.1c, at line 21, we defined a linear memory with a minimal size of 32-KiB. WebAssembly linear memory can also associate with zero or multiple *data* segments. Each *data* segment contains a constant evaluated expression, representing the initialization offset, and a sequence of bytes that the runtime environment will copy from. WebAssembly indirect tables are similar to linear memories, but they store function pointers instead of bytes. A *indirect table* has a type that consists of an upper and lower bound similar to *linear memory*, as well as a function type indicating the type of the function pointers allowed [5]. WebAssembly tables also introduce their initializer, *element* segments. The *element* segment is similar to the *data* segment, but it initializes function pointers instead of bytes.

[4]The upper bound is optional

[5]Currently, the function type must be `funcref` which is a union type of all possible function types.

Another difference between linear memories and indirect tables is that indirect tables are immutable after initialization to ensure the module's safety [6].

Linear memory Similar to asm.js, WebAssembly programs can access one or multiple *linear memories* [7]. The memory is unmanaged, and it is the program's responsibility to handle the layout correctly. The program can grow the *linear memory* if needed via the `memory.grow` instruction; however, the runtime environment is not obligated to increase the *linear memory*. The program can check the result of the command via the instruction's return value. Asm.js also allows the growth of the heap byte array. However, due to the limitations of JavaScript, this operation is usually quite expensive, as there is no efficient `realloc` algorithm provided in JavaScript, and it requires allocating a byte array with a larger capacity and copying byte-by-byte. WebAssembly specification does not impose requirements on the time complexity of growing the linear memory, yet it encourages any implementation to avoid copying. Unlike native heap memory, there is no alignment requirement on load-store instructions; i.e., load-store can start at any byte in the memory with the probable additional cost for unaligned access. However, there are boundary checks applied to the linear memory. Any out-of-bound access will result in a runtime panic. Additionally, WebAssembly specification requires any runtime environment implementation to zero-initialize the linear memory.

Indirect table Asm.js represents function pointers using first-class function values, thanks to JavaScript. However, in WebAssembly, every entity is referred to with indices representing references, and value types only consist of integral types and floating-point types [8]. Hence, we need something creative to implement the function pointers in WebAssembly. The solution utilizes one special instruction `call_indirect` and indirect tables. During module initialization, the runtime environment will initialize the indirect table according to the *element* section. Each `call_indirect` instruction associates with an in-

[6]This is subject to change in the reference type extension
[7]In the current version of WebAssembly, at most one linear memory is allowed within a single module
[8]WebAssembly may introduce more primitive value types in the future.

dex and an expecting type. The runtime environment will perform both a validity check on the index and a type check against the expecting type. Unlike the linear memory, the indirect table is not growable at runtime and is currently immutable once the initialization phase is complete. An indirect table does not limit the function pointers stored to be internal functions nor even WebAssembly functions. The function pointer can even be a host native function; many runtime environment implementations utilize this feature to register native call-back functions to WebAssembly modules.

Structured control flow Another WebAssembly's key feature is the structured control structure. Unlike the native binary and most of the bytecode representations that utilize labels and offsets, WebAssembly has structured control flow instructions and classifies them into three categories, *block*, *if* and *loop*, similar to asm.js. Each control flow instruction can optionally associate with a value type, representing the change on the operand stack once the control block exits [9]. A *block* control flow is perhaps the simplest structure. It introduces a label index to the context. The label is only referable within the *block* construct by indices. If a branch instruction refers to the block's label, the runtime environment will redirect the control flow as if it reaches the *block*'s end. An *if* control flow is similar to the *block* control flow with two significant differences. One is that it will implicitly consume a 32-bit integer from the stack and choose the branch accordingly. The other difference is that it can optionally have a *false* branch. If the *false* branch is missing, the runtime environment will redirect the control flow to reach the if's end, similar to the *block* control flow structure. The last control flow structure is *loop*. The only difference between the *loop* control structure and *block* structure is when a branch instruction refers to it. When a branch instruction refers to a *loop* block, the runtime environment will redirect the control flow to the *loop*'s beginning instead of the end. In the figure 2.1c, we present

[9]WebAssembly multivalue extension relaxes the requirement and allows structured control instruction to have a function type. If a control instruction associate with a function type, the parameter types refer to the value consumed from the operand stack and result types refer to the value added to the operand stack.

the *if* structure on line 5, and *loop* structure on line 7. The example does not contain a *block* structure, but there is no difference between it and a *loop* structure at the syntax level.

Generally speaking, WebAssembly's performance, compared to its native counterpart, varies significantly from test case to test case. On the browser side, WebAssembly can finish most test cases within 10% slower than the native version and all test cases within two times slower [8]. Another test shows similar results for most test cases, except one case is 2 times to 3.4 times slower than native, depending on the input size [12]. For generated code size, the community introduction paper claims 85.3% compare to native implementations. WebAssembly is not only successful in the field of Web-based applications. It also defines a portable format for distributing programs over the internet, similar to what we have seen in Java and its virtual machine. GraalVM now has its interpreter for WebAssembly modules, TruffleWasm [28], and can execute WebAssembly modules with impressive performance with only 4% slower than WebAssembly reference implementation in most of the cases, and even 4% faster in PolybenchC.

2.3 WebAssembly Extensions

In the previous section, we presented the core part of WebAssembly published by the community in late 2016 as a minimal viable product (MVP). Although the WebAssembly MVP is powerful enough to host most of the applications [20], there still exists room for improvement. These post-MVP proposals enhance the functionality of WebAssembly by introducing new instructions or modifying existing module constructs. For example, MVP WebAssembly has no support for exception handling. Thus, when compiling programs implemented in C++, users need to turn off the compiler's exception feature explicitly. The exception handling post-MVP extension addresses the problem by introducing a special `try` block, which enables user-defined stack unwinding. Most post-MVP extensions are still in the early stage of development and may merge into core WebAssembly in the future. This project implemented several post-MVP features such as integral value sign extension, non-trapping floating-point conversion, multivalue semantics, and fixed-width SIMD vector operation. In this section, we will quickly visit these post-MVP feature extensions.

Integral value sign extension MVP WebAssembly only has 32-bit and 64-bit integral values. However, many programming languages support integers with a smaller width. Thus, implementing short integral values in WebAssembly is quite awkward. To alleviate the problem, MVP WebAssembly has instructions that can perform load and store of 8-bit and 16-bit integers with signed or zero extension semantics. However, what if one already has a short integer on the stack and would like to perform a sign extension? Unfortunately, there are no immediate solutions. One possible work-around is to store the value to the linear memory and then sign extend with the load instruction's help, which is quite expensive. The sign extension proposal introduces new instructions that perform the sign extension for stack values. For example, `i32.extend8_s` consumes a 32-bit integer from the stack then performs the sign extension to the operand as if the operand is an 8-bit integer. The proposal also introduces similar instructions for 64-bit integers.

Non-trapping float-to-int conversion MVP WebAssembly offers floating-point-to-integer conversion with implicit range checks to fulfill the no-undefined behaviour design goal of the language. If the desired integer type cannot accurately represent the floating-point value, the runtime environment should trap. However, in most other languages, such as LLVM, the conversion yields an undefined result without trapping in such scenarios. Thus, if one wants to simulate the conversion between floating-point and integers faithfully, an `if` block with manual checks is usually required. This proposal introduces saturated value conversion to address the problem. If the desired integer type cannot represent the resulting number, the instruction employs saturated semantics. More specifically, if the floating-point value is more significant than the maximum representable value of the integer type, the maximum value is returned, and the same holds in the case of value underflow. This extension also lays the foundation of SIMD vector operations to achieve more hardware-like semantics, which we will see later in this section.

Multivalue The multivalue proposal focuses on two aspects of WebAssembly, the function return value and the types of structured-control-flow constructs. In MVP WebAssembly, the function can have at most one return value. The proposal generalizes the function type by allowing functions to return multiple return values. For structured-control-flow constructs, MVP WebAssembly requires that any instructions within the construct cannot consume stack values outside of the stack frame. Additionally, the construct can put at most one value onto the stack when it exists. One advantage of having such strict rules on structured-control-flow constructs is that the validation rule is trivial, and the runtime system can compute the stack height with minimal effort. However, this has its drawbacks. For example, this method causes the bloat of local variables. When entering a structured-control-flow construct, the program needs to push all the values it may need to the local variables, then load them back to the stack later, which is quite expensive. The multivalue proposal relaxes such constraints by allowing the control-flow-construct to have a function type. Function types' parameter types indicating the type of values that the construct will consume, and the result types hint at the type of values that will be pushed onto the stack.

SIMD vector operations Single-instruction-multiple-data (SIMD) is a powerful tool for implementing high-performance programs. Many modern compilers, such as GCC, have auto-vectorization analysis and transformation to automatically rewrite scalar codes in parallel form [23]. Before WebAssembly, many attempts have been made to implement SIMD operations over the internet, most notably, SIMD.js [10]. The design of the SIMD vector operation proposal is based on the design of SIMD.js. Currently, the proposal focuses on 128-bit vector operation, which is widely available on different hardware architectures such as SSE [27], and ARM Neon [11]. The proposal introduces a new value type `v128`

[10] `https://hacks.mozilla.org/2014/10/introducing-simd-js/`

representing a 128-bit vector. Note that the vector type does not contain any knowledge about the element type and how to interpret the lane, which is the number and the type of elements packed into a single vector, depends on the instruction. The SIMD vector operations proposal takes instructions from the intersection among different hardware architectures to ensure the module's portability. For example, `i32x4.add` will interpret both operands as packed 32-bit integers and perform lane-wise addition between them, while `f64x2.sqrt` will interpret its operand as a packed double-precision floating-point numbers. Most of the instructions are a direct generalization of the scalar operations in MVP WebAssembly. One notable difference is the floating-point conversion semantics. In MVP WebAssembly, the conversion will trap in the case of overflow or underflow. In contrast, in the SIMD vector proposal, packed floating-point value conversions follow similar semantics to those defined in the non-trapping float-to-int conversion proposal.

2.4 WebAssembly System Interface (WASI)

In the previous section, we introduce WebAssembly as a new format for delivering programs over the internet. The question then arises: can we push WebAssembly beyond the browser? On the other hand, if we want to compile the native program into WebAssembly, how do we translate operating-system-specific commands, such as file access? Taking a step further, how do we ensure the safety of the generated program? In the early days of development, Emscripten generates JavaScript glue code that mimics the operating system syscalls. However, this ad-hoc solution results in messy and nonportable code.

To address these problems, the WebAssembly community started the process of standardizing the system interface for modules [11]. The WASI interface design focuses on two aspects, portability and safety, following the WebAssembly design philosophy. The interface is still under active development at the time of thesis writing. In this project, we implement the interface functions only if they are needed while designing the backend

[11]WASI initial announcement:
`https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/`

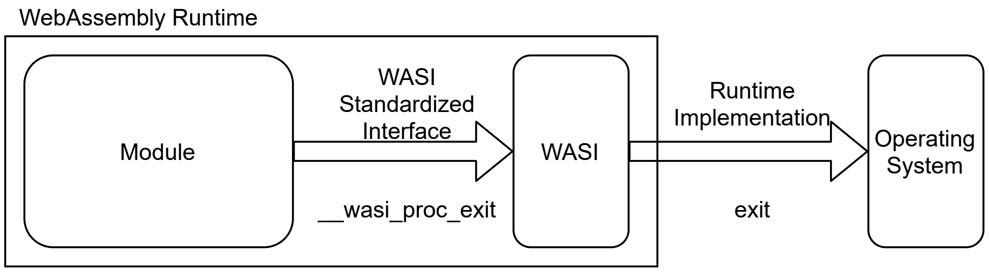


Figure 2.2: A illustration of WASI

library to be extensible. The official API documentation provides a detailed view on the design of the interface [12]. Figure 2.2 gives a general illustration of the relationship among the WebAssembly module, the runtime environment, and WASI. Here we will focus on several key points of the design.

WASI ABI model WASI classifies modules into two categories, *commands* and *reactors*. A *command* module has a single entry function, namely `_start` and all the other exported functions are hidden from the user. On the other hand, a *reactor* module has an optional initialization function named `_initialize`. If the initialization function is present, the runtime environment is obligated to invoke such a function before calling others. The runtime environment may invoke either the start or initializer function once during a module's lifetime. Additionally, every WASI-compatible model needs to export a linear memory under the name `_memory`, and all addresses referred by modules are offsets within this linear memory. Similarly, modules will also export an indirect table under the name `__indirect_function_table`. The runtime environment will pass function pointers through the indirect table. Additionally, WASI requires the runtime environment to provide all WASI API under module name `wasi_snapshot_preview1` [13].

Sandbox As we described above, WASI API follows WebAssembly's design philosophy, safety, performance, portability, and compactness. WASI modules execute under a

[12]WASI API documentation:
`https://github.com/WebAssembly/WASI/blob/main/phases/snapshot/docs.md`
[13]This will change in the future, as WASI is still in the standardization phase.

capability-based security system to ensure the safety of the host environment. The host runtime system will provide a sandboxed environment for each model. For example, for file system access, WASI standard library C works with a virtual file system for each module with the help of libpreopen [14] [15].

Non-invasive and extensible In our discussion above, one may notice that a WASI-compatible module is also a valid WebAssembly module on its own. WASI does not introduce new instructions or sections to the module; instead, it provides additional functionalities through imported external functions. The design of WASI is also highly extensible and split into separate modules. Currently, the WASI working group focuses on developing the core part that provides most of the POSIX interface, but it may add additional features in the future.

2.5 LLVM Compiler Infrastructure

The last section of this chapter briefly overviews the compiler pipeline design and LLVM compilation framework. Designing a robust and efficient compiler in terms of both generated code and compilation speed is challenging. The LLVM compiler framework [15] alleviates the problem by introducing a standardized intermediate representation (IR) between the compiler frontends and backends. Backend developers can target their analysis and transformations on the IR instead of specializing in different languages. On the other hand, frontend developers can translate the source language into the IR and expect the backend to support multiple target platforms with efficient code generation. Figure 2.3 illustrates the LLVM compilation pipeline. In this project, we are more interested in the frontend of the framework. The LLVM official documentation and tutorial provide full details of their intermediate representation [16]. Here we will only discuss several major

[14]libpreopen: `https://github.com/musec/libpreopen`
[15]In the more recent version of WASI libc, libpreopen is no longer required.
[16]LLVM Language Reference Manual: `https://llvm.org/docs/LangRef.html`

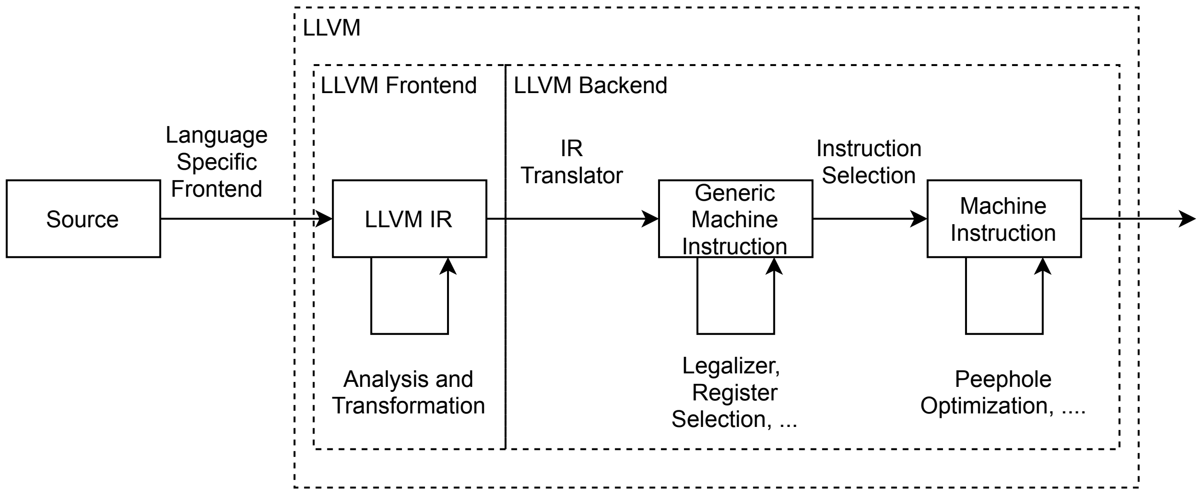

LLVM
LLVM Frontend
LLVM Backend
Language Specific Frontend
Source
LLVM IR
IR Translator
Generic Machine Instruction
Instruction Selection
Machine Instruction
Analysis and Transformation
Legalizer, Register Selection, ...
Peephole Optimization, ...

Figure 2.3: A illustration of LLVM compilation pipeline

differences between LLVM IR and WebAssembly that help understand the thesis. We also provide an implementation of Adler-32 hashing in LLVM IR generated with Clang in figure 2.4.

Register-based IR against stack-based IR In WebAssembly, all instructions operate over an implicitly declared stack. For example, in figure 2.1c at line 20, a 32-bit integer constant instruction, `i32.const`, will push the constant value on the stack, and a 32-bit add instruction, `i32.add` will pop two values off the stack as left-hand-side and right-hand-side operands accordingly, then push the sum onto the stack. On the other hand, LLVM utilizes a register-based IR, which is more similar to what one would expect on a native machine. In figure 2.4, each value for example, `%0`, `%1`, etc is a virtual register. Later in the backend, the register allocation pass will map the virtual registers into physical registers using register allocation algorithms.

Control flow, basic block, and ϕ instruction As we saw in previous sections, WebAssembly has specialized instructions to manage the program's control flow. On the other hand, LLVM took a more traditional approach to the problem. In 1991, researchers from IBM introduced *static single assignment* (SSA) form to ease the difficulty of writing program analysis and transform passes [2]. In SSA, each value has its definition exactly once, and

```
1  define i32 @adler32(i8* %0, i64 %1) {
2  2:
3    %3 = icmp eq i64 %1, 0
4    br i1 %3, label %6, label %10
5
6  4:                                                      ; preds = %10
7    %5 = shl nuw i32 %20, 16
8    br label %6
9
10 6:                                                      ; preds = %4, %2
11   %7 = phi i32 [ 1, %2 ], [ %18, %4 ]
12   %8 = phi i32 [ 0, %2 ], [ %5, %4 ]
13   %9 = or i32 %8, %7
14   ret i32 %9
15
16 10:                                                     ; preds = %2, %10
17   %11 = phi i64 [ %21, %10 ], [ 0, %2 ]
18   %12 = phi i32 [ %20, %10 ], [ 0, %2 ]
19   %13 = phi i32 [ %18, %10 ], [ 1, %2 ]
20   %14 = getelementptr inbounds i8, i8* %0, i64 %11
21   %15 = load i8, i8* %14, align 1
22   %16 = zext i8 %15 to i32
23   %17 = add nuw nsw i32 %13, %16
24   %18 = urem i32 %17, 65521
25   %19 = add nuw nsw i32 %18, %12
26   %20 = urem i32 %19, 65521
27   %21 = add nuw i64 %11, 1
28   %22 = icmp eq i64 %21, %1
29   br i1 %22, label %4, label %10
30 }
```

Figure 2.4: Adler 32 in LLVM

hence, the use-definition chain (UD chain) is trivial to compute. The UD chain presents the relationship between variable declarations and variable-uses in a graph. It helps the analysis pass to efficiently pinpoint the variables and identify if the variable declaration is necessary. However, in most programs, this information needs to be merged from different control-flows; for example, in a for-loop, the loop counter may be defined in the loop initialization and on each loop iteration. The SSA introduces a special instruction, ϕ instruction, explicitly marking the merge of definitions from different execution paths. LLVM adopts this design principle in its intermediate representation. In figure 2.4 we have multiple ϕ instructions. For example, at line 11 and 12, value $\%7$ and $\%8$ represent a and b accordingly. We know that a and b initialized to 0 and 1 upon entry and updated on each iteration from our C implementation. In the generated LLVM IR, these merges induce ϕ instructions. For a (%7), if the control flow is from the beginning of the function, we set its value to 1, and on the other hand, if the control flow is from the loop iteration, we update its value accordingly. The different paths inducing a ϕ instruction are indicated by basic block numbers. A basic lock groups the maximum number of instructions without control flow transfer. At line 11, we see the ϕ instruction merges the definition coming from the $\%2$ which is the entry block and $\%4$. Additionally, ϕ instructions must appear before any other instructions within the same basic block, as they model the merging of values and do not have any execution semantics.

Memory and load-store instruction The last significant difference between WebAssembly and LLVM IR is on the memory and its related instructions. As we discussed earlier, a WebAssembly module can have access to multiple linear memories [17]. One might confuse WebAssembly's linear memory with the concept of address space in LLVM IR. LLVM IR associates each address with an integer value, namely, the address space. However, unlike linear memory in WebAssembly, which has no difference between one and another, the LLVM backend interprets the address space differently for various architectures. For example, in the PTX backend, a backend target for Nvidia GPUs, the implicit address space 0 refers to traditional main RAM, and address space 4 represents the address shared by both main RAM and GPU RAM [18]. For most architectures, the implicit address space is the only address space available to the programmer. Another difference between WebAssembly and LLVM IR is in the design of load-store instructions. Load store instructions in both languages have an attribute of alignment. However, LLVM IR interprets this attribute differently from WebAssembly. In WebAssembly, the alignment attribute acts as a hint to the runtime environment. If the alignment hint is unsuitable, the runtime environment should still proceed under a possible penalty in the performance. However, in LLVM IR, the alignment attribute is a requirement. Any memory access that violates

[17] In the current version of WebAssembly, only one linear memory is allowed per module

[18] An introduction for PTX backend:
`https://llvm.org/devmtg/2011-11/Holewinski_PTXBackend.pdf`

the alignment attribute will result in undefined behaviour, usually a runtime panic. A load-store instruction in LLVM IR with alignment set to one will never fail. However, it will be significantly less efficient as the backend will likely generate byte-wise load and concatenation instructions.

We visited some of the background information that helps with understanding the thesis in this chapter. The next chapter will start from the beginning of the system implementation, the WebAssembly parsing and validation frontend.

Chapter 3

Frontend

This chapter describes the frontend of SableWasm. The frontend consists of two parts, the bytecode parser and the validation pass. WebAssembly is a continuously evolving language, and its community might add new instructions in the future. Hence, the design of the parser and the bytecode validation phase closely follows WebAssembly's specification and is modular to ensure the framework's extensibility. Additional functionalities are provided via a read-only view of the module structure. The design of the parser and the validation phase focuses on performance, both in execution time and memory footprint.

3.1 Bytecode Parser

One of WebAssembly's binary format design goals is to be simple to parse. Although open-source bytecode parsing and validation libraries have become available at this point, such as WABT [1] provided by the WebAssembly community, there was no suitable library at the time when the project starts. Thus, for SableWasm, we implemented our bytecode parsing frontend instead. The bytecode parser consists of three components: the byte-source reader, WebAssembly bytecode parser and the parser delegate. This section will

[1]WebAssembly Binary Toolkit: `https://github.com/WebAssembly/wabt.git`

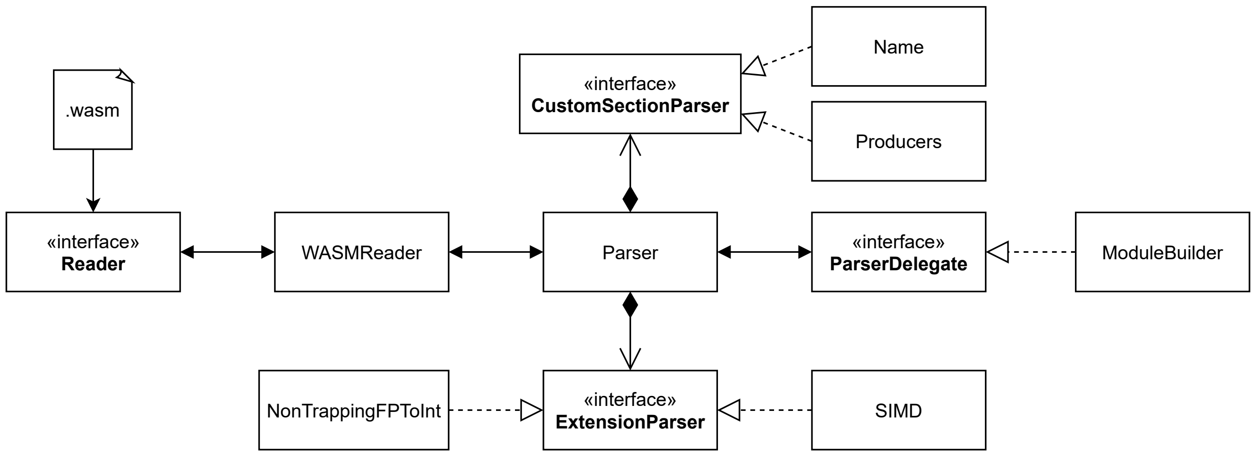


Figure 3.1: SableWasm parser

give a brief description of each component, and figure 3.1 presents a general illustration of the parser design.

Byte-source reader The byte-source reader consists of two parts, the byte-buffer reader and the WebAssembly reader. The byte-buffer reader provides essential functionalities such as reading and skipping ahead. Additionally, the byte-buffer reader also needs to support rewind and enforce an end-of-stream barrier. Any out of bound access, either beyond the barrier or if the byte stream is exhausted, will signal via exceptions. On the other hand, the WebAssembly reader provides a richer interface to the parser, such as the ability to decode LEB-128 encoded integers and parse WebAssembly value types. The WebAssembly reader is also responsible for validating the result before passing it to the parser. In the case where the result is invalid, the reader throws exceptions similar to the byte-buffer reader.

WebAssembly parser WebAssembly parser is the kernel part of the parsing framework. As we discussed earlier in this chapter, one of the primary design goals of the framework is its extensibility. Hence, the SableWasm parser is modular and consists of three parts: the parser core, the custom section parser and the instruction extension parser. The grammar for the WebAssembly binary representation is quite simple, and therefore, the parser core implements a simple top-down recursive descent parser with a single byte look-ahead.

Custom sections are a special section defined in the WebAssembly standard. They are essentially a binary data chunk tagged with a string name. How to interpret the binary data can be different in each case. These custom sections can either be standardized by the community or defined as specific to a toolchain. In this project, we implement two custom sections standardized by the WebAssembly working group, namely *Name* section and *Producer* section. The *Name* section gives human-readable names to functions and their local variables that help with program debugging. The specification does not require these names to be the same as the import or export names. There is no direct support for more detailed debug information encoding in WebAssembly at the time of thesis writing; however, extensions are working on this problem, such as DWARF for WebAssembly [2]. The *Producer* section is relatively simple. It only encodes information about the toolchain that generates the module, such as the toolchain name and version. All custom section parsers in SableWasm are derived from the base class `CustomSection`. The parser core will dispatch the binary chunk to the corresponding custom section parser based on the name tag. Each custom section parser manages its results and does not communicate to the parser delegate directly.

Instruction extension parsers focus on another different aspect of the WebAssembly module. In the background section, we have visited several extensions that merged with the WebAssembly specification. A quick reminder, WebAssembly extensions can insert or modify the instructions defined in the minimum-viable-product (MVP) specification. The SableWasm WebAssembly parser employs instruction extension parsers to address this problem. When the parser intends to parse an instruction, it will iterate over all its instruction extension parsers in a chained manner. If the instruction opcode is not recognized by any registered instruction extension parser nor in the minimum-viable-product specification, the parser will signal the error by throwing an exception. An instruction

[2]DWARF for WebAssembly: `https://yurydelendik.github.io/webassembly-dwarf/`

extension parser can also override the default behaviour for MVP instructions by handling instructions early, though, in the current version of WebAssembly, no extensions modify the semantics of these instructions. In this project, we implement two instruction extension parsers, the non-trapping-float-to-int conversion parser and SIMD parser, which handles the instructions introduced by the extensions as their names suggest.

Parser delegate The last part of the SableWasm WebAssembly bytecode parser is the parser delegate. The parser delegate and the parser core directly implement the typical delegation pattern seen in many other projects, separating the parsing logic from the heavy lifting of module construction. One can implement a validation pass at this level without module construction. However, in this project, we implement our bytecode validation pass after the module construction, giving space for further projects focusing on bytecode-level transformation. We will discuss the implementation of such a validation pass later in the chapter.

In this section, we gave a brief overview of the parser framework introduced in SableWasm. In the next section, we will discuss the WebAssembly bytecode representation used in the project and several techniques to improve the performance and ensure flexibility.

3.2 WebAssembly Bytecode Representation

The WebAssembly specification provides compact representations in both binary and text formats [8], and might subject to change in the future. In SableWasm, we implement our bytecode representation as close to the specification as possible. Hence, in the future, if the community alters the specification, we can straightforwardly update the bytecode representation without introducing extra complexity. In figure 3.2, we present an illustration of the bytecode representation used in SableWasm. The WebAssembly bytecode representation in SableWasm consists of three layers: the module, the entities, and the

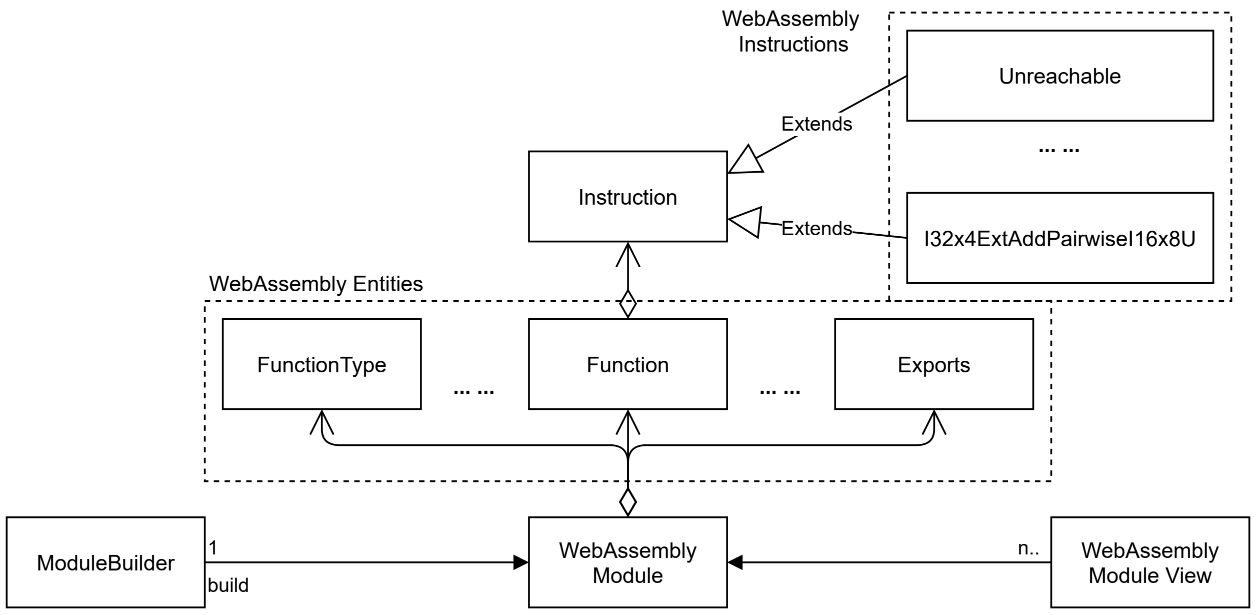


Figure 3.2: SableWasm bytecode representation

instructions, which we will discuss in detail later in this section. Compared to the representation given in the WebAssembly specification, the only difference we have is the `Function`. The standard WebAssembly bytecode representation splits the function section into two different sections, *function* and *code*, to achieve its one-pass validation goal. The *function* section contains the type of all functions defined in the module, similar to function declarations in other programming languages. Later in the module, the *code* section defines them. On the other hand, in SableWasm, we merge these two sections into a single `Function` object. A `Function` in SableWasm bytecode representation contains both its type and body definition.

WebAssembly module view The WebAssembly module structure only serves as a storage container for the bytecode representation and itself does not provide an interface to the user, except by retrieving entities by index. Additionally, the WebAssembly standard binary format focuses more on compactness instead of usability, which leads to complexity when retrieving the information. For example, to avoid duplication, the function types are stored in their own section, namely, the *type* section. Later in the module, any reference to the type becomes indices within this section. Another example is entity indices. The WebAssembly specification requires that every import entry in the *import* section implicitly introduces an index in its corresponding class. These indices should come before any definition introduced in the module. These two rules suggest that to retrieve an entity by index, one should first iterate over all the imports and then locate the entity accordingly, which is a relatively expansive operation. To address these problems, we implement a read-only view of the WebAssembly module that caches the indices and provides additional features.

Instructions SableWasm takes a traditional 'abstract syntax tree' approach to bytecode instruction representation. The frontend represents each instruction using a corresponding class derived from a common base class, namely `Instruction`, and an expression with a vector of instruction pointers. One observation is that the heap memory usage grows rapidly, as each instruction requires a unique heap-allocated object, which is not optimal. In WebAssembly, instructions operate over an implicitly defined stack, and for most of the operations, there are no operands attached to them. For example, `F32x4Nearest` has no operand; it will pop a value from the stack, treat it as a vector of packed single-precision floating-point values, round them to the nearest integer, and finally push the result back to the stack. From the bytecode representation point of view, there is no difference between multiple instances of the same instruction. Hence, we use pointers that point to object singletons to represent instructions without operands to reduce memory consumption. However, this introduces a problem in distinguishing a pointer that points to an object from one referring to an actual heap-allocated object which requires memory deallocation. To address this problem, we use tagged pointers. For a non-heap allocated singleton object, we tag the least significant bit in the pointer with zero; and on the other hand, we tag that of a heap-allocated object pointer with one. Later, we only need to examine the pointer's least significant bit within the destructor and deallocate memory when needed. With tagged pointer techniques, we can significantly

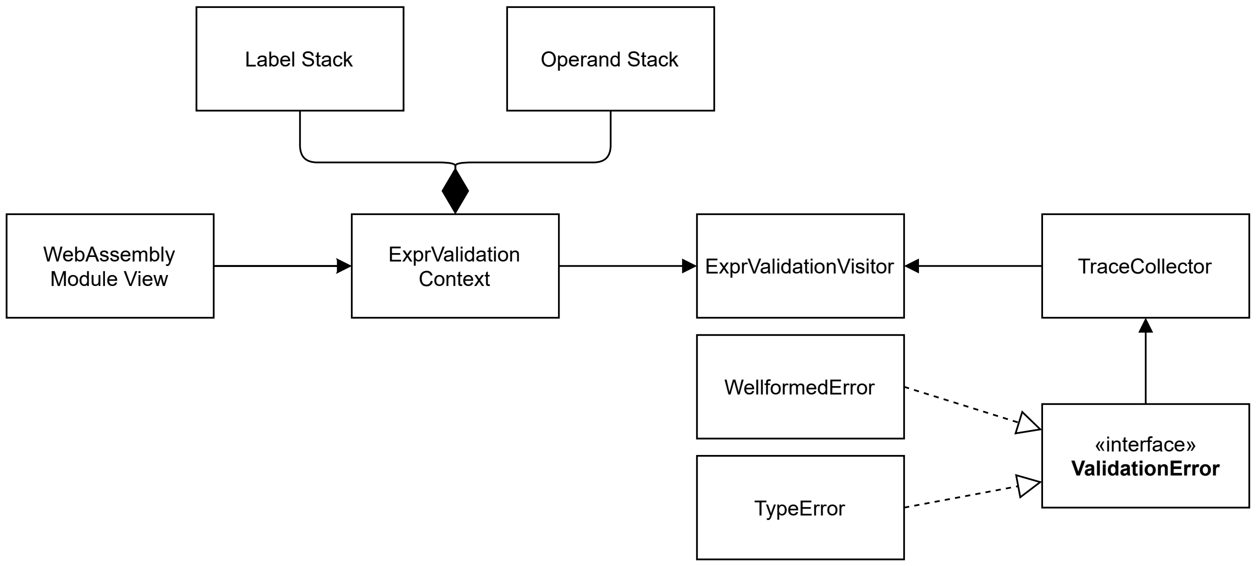


Figure 3.3: SableWasm validation pass

reduce the memory needed to store the bytecode representations while maintaining their polymorphic nature. As less memory allocation is needed, we also observe performance improvement in terms of execution time, which we will later see in this chapter.

This section gave an overview of the bytecode representation used in SableWasm and several techniques to improve its performance. In the next section, we will move to the validation pass implementation in SableWasm.

3.3 WebAssembly Bytecode Validation

The WebAssembly specification defines detailed static validation rules both for well-formedness and type-soundness. Similar to the parser framework, we implement our validation pass as close to the specification as possible. If there are changes to the specification in the future, we can adopt them with minimal effort. The validation pass implementation consists of three parts: the validation context, the validation visitor, and the trace collector as illustrated in figure 3.3. Later in the section, we will give a brief introduction to each of the components. The detailed validation rule, both well-formedness and type-soundness, are listed in the WebAssembly introduction paper [8], and a separate paper that focuses on validation [35]. Note that these two papers only present the validation rules for the minimum viable product (MVP) WebAssembly and each extension may modify the specification. The additional validation rules introduced by the extensions adopted by SableWasm are relatively trivial. Hence, we will not give a detailed description here, and one should consult the extension proposals for detailed information.

Validation context The validation context implements the context defined in the WebAssembly specification. It provides an easy way for the validation visitor to access the WebAssembly entities' declarations. The validation context itself does not perform any error signalling. The validation context also manages the operand stack and the label stack. The operand stack stores the type information gathered from the instruction within the expression while the label stack keeps track of the signatures of the control flow structures. The operand stack also records the requirements generated from type variable sequences and checks if there are contradictions among them.

Validation visitor The validation visitor is the core driver part of the validation pass. It implements all the validation rules for each instruction and derives from the visitor template. In the current state of implementation, the design of the validation visitor is not modular, and hence, if additional instructions are added to the project later, direct modification is required. The validation visitor also handles all the error signalling. However, it does not construct the error objects by itself. The task is deferred to the trace collector. For most of the instructions, the validation rule is quite simple, involves only popping and pushing values to the operand stack. Currently, the validation visitor will stop at the first error it encounters due to the WebAssembly validation rules' design.

Trace collector The last part of the validation pass implementation is the trace collector. It locates the position of the instruction that currently undergoes validation. It first keeps track of the section where the instruction lives with an enumeration and uses a stack to track how to locate it. Every time the validation visitor enters a nested construct such as `if`, `block` or `loop`, it pushes the construct to the site stack and pops when a nested expression finishes validation. If the validation visitor locates an error, the trace collector will build the error by moving the trace stack into the error object.

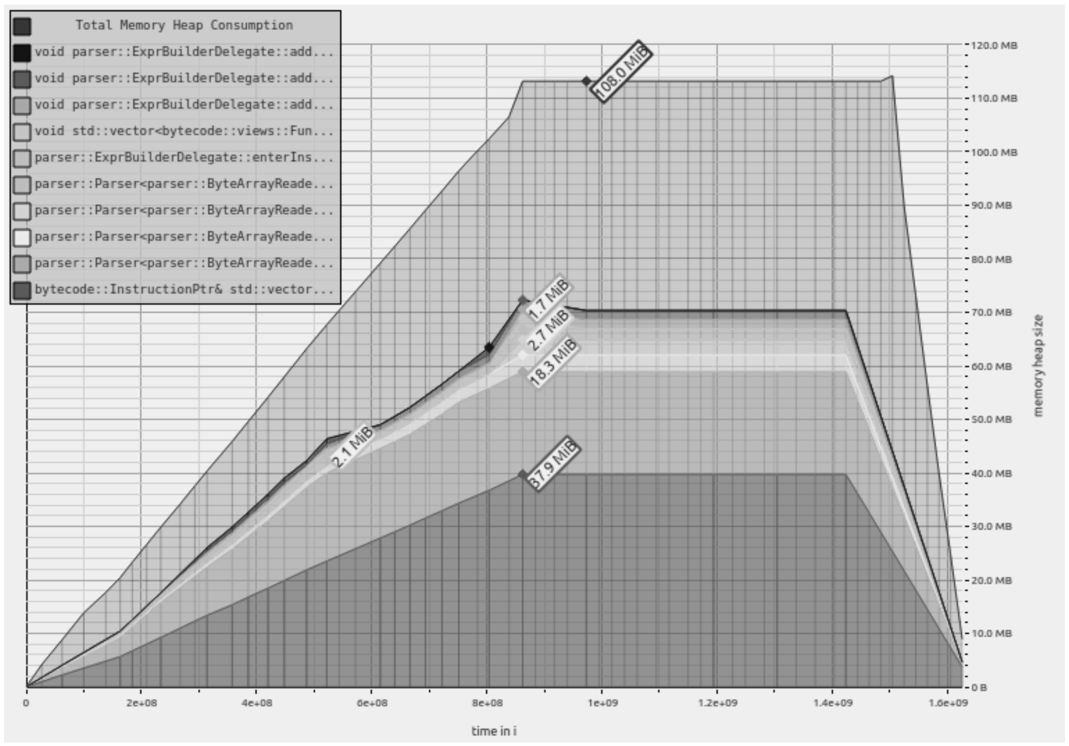
In this section, we presented the validation pass implementation in SableWasm. In the next section, we will perform some experiments to evaluate the system's performance in terms of both execution speed and memory footprint.

3.4 Performance Evaluation

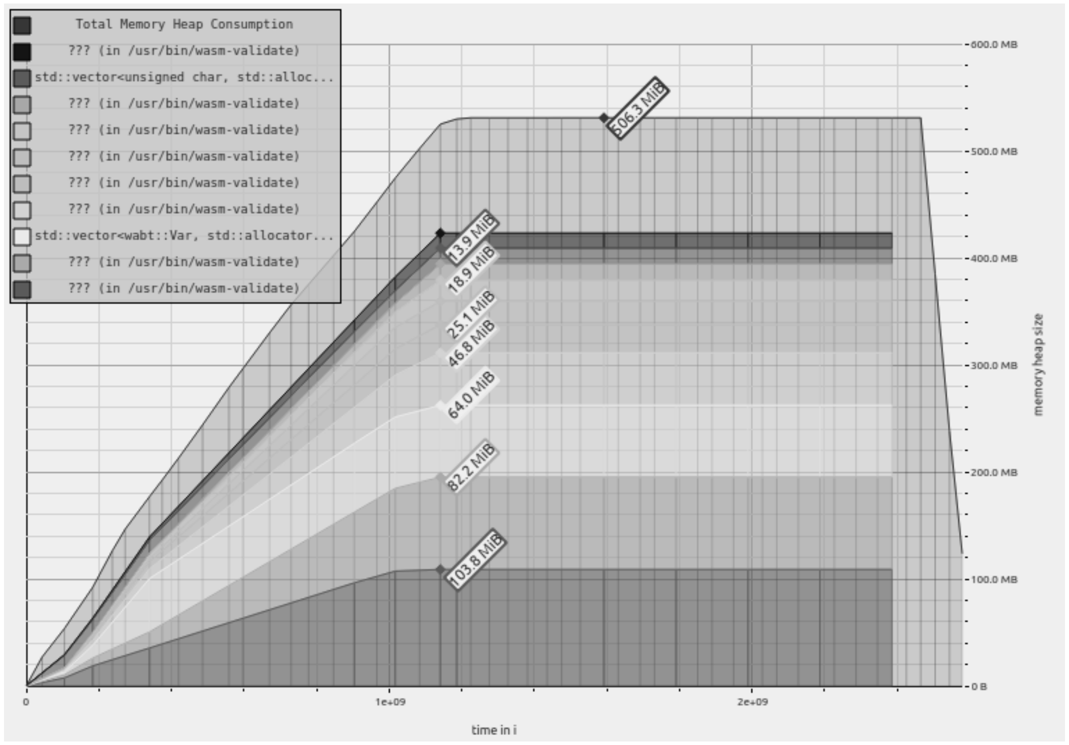
This section presents the performance benchmark comparison between the SableWasm parser frontend and the WebAssembly binary toolkit (WABT) offered by the WebAssembly community group. The benchmarks focus on both the execution time and memory footprint. We will first present the benchmark setup and then evaluate the data collected from the experiment.

Benchmark setup Experiments were performed on a server with a six-core Intel Core processor at 3.7 GHz standard clock frequency and an L3 cache of 12MiB. The server runs Ubuntu 18.04 with Linux kernel version 4.15.0 and 32GiB of memory. For the benchmark subject, we choose Pyodide [3], a WebAssembly implementation of Python 3.8. The project has reasonable complexity, and the size of the module is quite significant, which reduces the measurement errors during the benchmark. The WebAssembly binary toolkit (WABT) we used during the benchmark is version 1.0.23, and we perform module validation with its `wasm-validate` command. This command should parse the WebAssembly module, construct an internal bytecode representation, and finally perform a validation pass over it. This procedure is similar to what we used in the SableWasm frontend. For the execution speed experiment, we time the execution speed with the shell builtin `time` command. We perform ten runs for each implementation in total and then compute the

[3]Pyodide project: `https://github.com/pyodide/pyodide`



(a) SableWasm



(b) WABT

Figure 3.4: Frontend memory footprint comparison

average execution speed. For memory footprint, we use the Massif tool provided by Valgrind [22]. Massif is a heap profiler that collects heap memory usage throughout a program lifetime, which we will use to compare two implementations.

Benchmark result We will go over the finding over the execution speed first and then analyze the memory footprint. Table 3.1 gives the result of the execution speed benchmark. The data is relatively consistent over a total of ten runs, where the SableWasm frontend achieves around 1.5x to 1.7x speedup compared to `wasm-validated` provided by WABT. On average, the SableWasm frontend is 1.6x times faster than WABT's implementation.

Run	SableWasm	WABT	Speedup
Run #1	0.311	0.523	1.682
Run #2	0.308	0.511	1.659
Run #3	0.307	0.511	1.664
Run #4	0.329	0.513	1.559
Run #5	0.336	0.520	1.548
Run #6	0.317	0.534	1.685
Run #7	0.315	0.518	1.644
Run #8	0.335	0.522	1.558
Run #9	0.311	0.513	1.650
Run #10	0.335	0.574	1.713
Average	0.320	0.524	1.635

Table 3.1: Frontend execution speed comparison

On the topic of memory footprint, figure 3.4a and figure 3.4b shows the memory consumption trace for SableWasm and WABT, respectively. As we can see from the figures, SableWasm consumes approximately 108MiB at peak while WABT uses around 506MiB, suggesting a 4.6x reduction in memory footprint. More specifically, SableWasm spends about 38MiB for bytecode representation, and WABT takes roughly 64MiB, indicating a 1.7x reduction for bytecode representation only. Note that in SableWasm's heap memory trace, we can accurately determine the memory consumption of bytecode representation using debug symbols. However, this is not the case for WABT, where the debug information is stripped from the executable. In this experiment, we estimate the memory consumption for bytecode representation via the vector that contains the opcode.

Overall the SableWasm frontend implementation performs better than WABT in terms of both execution speed and memory footprint. Of course, SableWasm is a static compiler that will only perform parsing and validation at compile-time and does not affect the overall execution time for emitted executables. However, in the future, if one would like to implement a just-in-time (JIT) style compiler, the SableWasm parsing and validation frontend can effectively improve the response time of the JIT compiler.

We visited the design and implementation of the SableWasm frontend, which takes care of parsing, validating and constructing a bytecode representation in this chapter. In the next chapter, we will move to the next phase in the SableWasm compilation pipeline, the middle-level intermediate representation.

Chapter 4

Middle-level Intermediate Representation

This chapter describes SableWasm's middle-level intermediate representation (MIR), which has a critical role in the entire compilation pipeline. The MIR acts as a middle ground between the WebAssembly bytecode frontend and various possible backends. Currently, SableWasm only implements one backend that utilizes the LLVM compilation framework, but adding more backend support should not require significant modification on the MIR. It also implements an analysis and transformation framework where we perform several optimizations over the MIR. We will first go over the overall design of the MIR, and later move to the translation rules and analysis framework in the chapter 5.

In the previous chapters, we covered the design of WebAssembly bytecode. A quick reminder, WebAssembly is a stack-based intermediate representation (IR) where all instructions operate over an implicitly declared operand stack. There are several advantages of a stack-based IR. Perhaps the most important one is its portability. A stack-based IR makes fewer assumptions on the machine than a register-based one. One can even provide an implementation for a hypothetical device with only one register. Another advantage is the code size. Experiments show that, in general, a stack-based IR is smaller in size than its corresponding registered version [30]. When designing a binary format that ships executables over the internet, the stack-based IR seems to be a better choice for WebAssembly.

Nevertheless, there are no silver bullets: a stack-based IR design also has its drawbacks. One of them is the difficulty faced when performing code analysis and transformation over the module. As for each instruction, its operands implicitly come from the stack; the value use-definition relationship between instructions is not apparent to the analysis, and recovering such connection between instructions from the IR is not a trivial task.

On the other hand, we have the register-based intermediate representation, commonly abstracted to assume an infinite number of registers and requiring a register allocation algorithm to map them to actual, physical registers. For each instruction in register-based IR, it has its operand encoded in the instruction. Hence, the use-definition relationship will become explicit to the analysis and transformation.

The main design goal for SableWasm MIR is to provide an analysis platform for the entire compiler system. Thus, we implement our MIR as an infinite register machine. We also take a traditional approach in various other aspects. For example, instead of using the structured control flow similar to what WebAssembly offers, we use *control-flow graphs* (CFGs) to represent the relationship between basic blocks. The SableWasm MIR is also in *single static assignment* (SSA) form [2], as covered in the background chapter. The design for instruction and module-level entities in SableWasm MIR is quite similar to what WebAssembly instruction offers. One can view the SableWasm MIR as a mixture of the target LLVM intermediate representation and the source WebAssembly bytecode. We also adopt several design features from LLVM IR into MIR, such as automatically managed use-site lists, which provide each AST node with an efficient way to access their use sites. In SableWasm MIR, all elements are derived from the base class `ASTNode` which implements these features that are helpful later in MIR analysis and transformation.

```
@export memory
memory %memory:0 : {min 2}

table %table:0 : {min 1, max 1} funcref

global %global:0 : var i32 = i32 66560

function %fibonacci : [i32] -> [i32] {
{(arg)%0:i32}
#pred = {}
%entry:
  %1 = local.get (arg)%0
  br.table %BB:1 0:%BB:2 1:%BB:0

#pred = {%entry}
%BB:0:
  %2 = const i32 1
  br %exit

#pred = {%entry}
%BB:1:
  %3 = local.get (arg)%0
  %4 = const i32 -1
  %5 = int.add %3 %4
  %6 = call %fibonacci(%5)
  %7 = local.get (arg)%0
  %8 = const i32 -2
  %9 = int.add %7 %8
  %10 = call %fibonacci(%9)
  %11 = int.add %6 %10
  local.set (arg)%0 %11
  br %BB:2

#pred = {%BB:1, %entry}
%BB:2:
  %12 = local.get (arg)%0
  br %exit

#pred = {%BB:2, %BB:0}
%exit:
  %13 = phi i32 [%2, %BB:0] [%12, %BB:2]
  ret %13

}
```

Figure 4.1: Fibonacci in translated SableWasm MIR

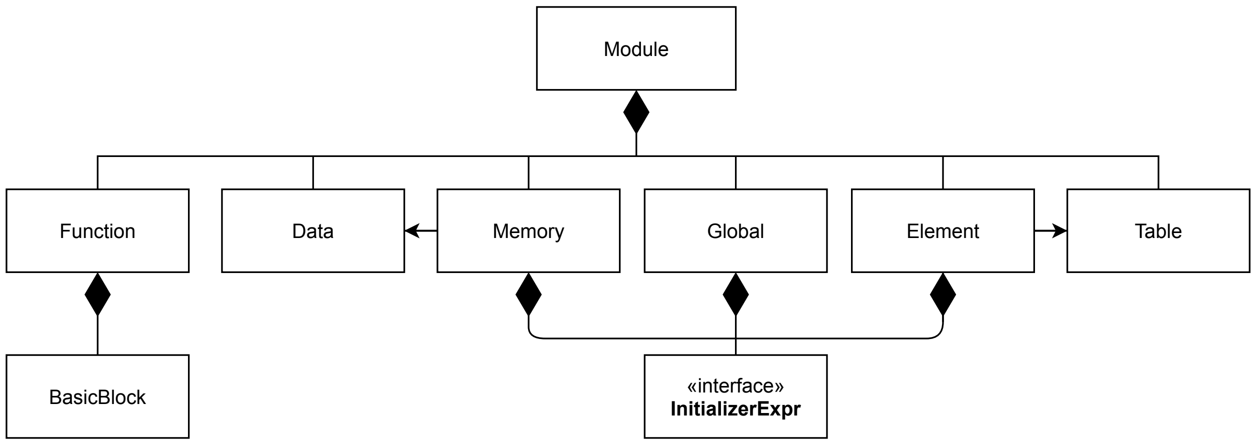


Figure 4.2: SableWasm MIR Module-level entities

Figure 4.1 shows a simple function that calculates Fibonacci numbers with a recursive method in SableWasm MIR. With the help of the figure, we will go through the detailed design of SableWasm later in the chapter. We will first present the module-level entity and their initializer expressions, such as functions, then move to the design of each instruction defined in MIR.

4.1 MIR Module Entities

SableWasm module-level entities are the top-level elements in a translation module. They are direct implementations of the WebAssembly module entities defined in the specification. Figure 4.2 presents a general illustration of the SableWasm module-level entities. In this section, we will cover the design of each entity and compare it with its WebAssembly correspondent. All SableWasm module-level entities can optionally have import and export annotates, except `data` and `element`. These annotations correspond to the import and export entries defined in the WebAssembly specification.

Function In figure 4.1, we have a function definition at line 8. A function declaration in SableWasm provides information about the type, local variables, and name. A function definition should satisfy all the function declaration requirements and, in addition, provide a function body using basic blocks. The design of the function declaration and definition in SableWasm is quite similar to that of WebAssembly. The only major difference is how to represent the function body. We will come back to this in the later sections within this chapter when we discuses the design of SableWasm MIR instructions.

Global SableWasm's global variable declaration and definition follow the design in WebAssembly. In SableWasm, we relax several of the constraints defined in the WebAssembly specification and its extensions. In the SIMD extension proposal, the 128-bit vector type, `v128`, is only suitable within the function body. There is no direct way to pass a vector value to the host environment, as there is a lack of standard representation for 128-bit packed vectors in JavaScript [1]. In SableWasm, we treat all primitive types uniformly. Thus, a global variable can contain an integral value, a floating-point value, or even a packed SIMD vector. The type for the SableWasm MIR global variable follows the specification in WebAssembly; it consists of a value type and a constness modifier. In figure 4.1, we have a global variable definition at line 6, which introduces a mutable 32-bit integral value. All global variable definitions in SableWasm must provide a value initialization via an initializer expression. In SableWasm MIR, all initialization expressions are constant expressions, meaning that the host system can deduce the resulting values at the module initialization phase. At runtime, the host system will first evaluate these expressions and then initialize the global variables accordingly. We will come back to the initialization expressions in detail later in this chapter.

Memory and Data Memory and Data are implementations of the WebAssembly linear memory and its initializer, respectively. One might think that there is no need to separate the memory initializer from the memory entity definition, as in WebAssembly specification, all data section entries must provide a valid linear memory index. In the early version of SableWasm, we indeed adopt such implementation. However, this approach might be subject to a significant change in an extension that might soon merge to the We-

[1]This might subject to change in the future.

bAssembly specification. The WebAssembly bulk memory operation extension proposal [2] introduces new instructions, such as `memory.fill` that directly refers to a data section segment. Moreover, the proposal relaxes the constraints on the linear memory index. Now the index can behave as a flag indicating whether the data segment itself is active or not and no longer serves as a linear memory index. Hence, to make our framework 'futureproof', we separate linear memory declarations from their initializers. Figure 4.1 presents a linear memory definition at line 2. SableWasm memory entities also adopt WebAssembly's linear memory type. The type consists of a pair of unsigned integers, indicating the lower bound and upper bound of the memory size in WebAssembly pages. The example above defines a memory with a minimal size of 2 pages, 128KiB, and exports it under the name 'memory'. It, however, does not provide any example for data initializers, although they are quite easy to understand: a data initializer is essentially a binary chunk with an initialization offset, and is semantically equivalent to a data section entry in an ELF file.

Table and Element SableWasm's table and element entity implement the indirect table and its initializer, namely element segment, accordingly. They follow the same principle as the memory and data entity in the previous section. Currently, like a data segment entry, WebAssembly's element section entry must refer to a valid indirect table via an index. In the future, this may also subject to change. The WebAssembly reference types extension proposal [3] introduces instructions such as `table.fill` that are able to have direct access to element segment initializers. `table.fill` instruction is similar to `memory.fill` defined in the bulk memory operation extension. It will copy a sequence of compile-time defined function pointers into an indirect table at runtime. Thus, when we design our table entity, we also split the declarations from their initializers. The type for table entity is the same as the table type in WebAssembly. It consists of a pair of un-

[2]WebAssembly bulk memory operations:
`https://github.com/WebAssembly/bulk-memory-operations`

[3]WebAssembly reference types: `https://github.com/WebAssembly/reference-types`

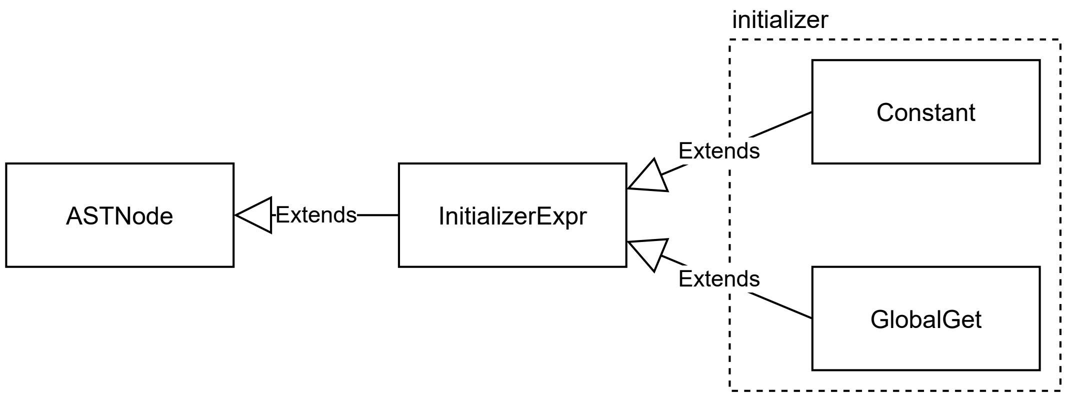


Figure 4.3: SableWasm MIR Initializer Expression

signed integers, indicating the lower bound and upper bound for the number of function pointers stored in the indirect table. In SableWasm MIR, we treat memory entities and table entities as black boxes, and its concrete implementation is deferred to the backend. In the example shown in figure 4.1, the module defines a table entity at line 4 that stores exactly one function pointer. Note that the table entity does not require users to initialize the value for all entries. The table entity default initializes all entries to null pointers.

In this section, we covered the design for module-level entities in SableWasm. They are pretty similar to the those defined in the WebAssembly specification. In the next section, we will move the design of SableWasm initialization expressions.

4.2 MIR Initializer Expressions

WebAssembly defines a particular form of expression for initialization, namely constant expressions. They can appear in three locations in the current specification. First, global variables declaration can contain constant expression as their initialization values. Additionally, data section entries and element section entries can have constant expressions as the offsets for their initialization payload. In SableWasm MIR, we define initializer expressions that act similar to what constant expressions do in WebAssembly. Figure 4.3 gives a general illustration about SableWasm MIR initializer expressions. The initializer expressions are quite simple. In the current WebAssembly and SableWasm, an initializer expression can be either a constant value or refer to an imported global via `GlobalGet` instruction. Hence, in principle, currently, a SableWasm MIR initializer expression is essentially a single instruction. In the future, one may generalize such constraints by allowing more complex constructs in initializer expressions.

Constant The `Constant` instruction represents a single constant value for the initializer expression. In WebAssembly, a constant value can be one of the following: a 32-bit or 64-bit integer, a floating-pointer number, or a 128-bit SIMD vector [4], and the specification encodes the type within the instruction opcode. Hence, there are multiple instructions in WebAssembly to introduce a constant. In SableWasm, we do not encode the type into the opcode, and `Constant` instruction is the only instruction that takes care of the task. In figure 4.1, we have a constant initializer at line 6 that initializes the value of the global to a 32-bit integer with a value that equals 66560. When querying the type of a `Constant` instruction, SableWasm will infer it according to its payload constant.

GlobalGet The `GlobalGet` instruction is exactly same as the WebAssembly's `global.get` in terms of execution semantics. The WebAssembly specification allows any initializer expression to refer to an imported [5] global value. As these values are initialized before entering the module, reading their value is always valid during module initialization. The example in figure 4.1 does not provide an example of `GlobalGet` as an initializer expression, as they are less frequently used compared to constant initializer expression, especially for global values. However, in some ABI implementations, data section entries and element section entries require reading from global values serving as base pointers. SableWasm also infer the type for `GlobalGet` initializer expression in a similar fashion as `Constant`. In this case, the type of instruction is the same as the referred global variable without the 'constant' modifier.

[4]With WebAssembly SIMD128 extension

[5]This might subject to change in the future version of WebAssembly

In this section, we covered the design and implementation of initializer expressions in SableWasm. They are pretty simple in the current design. We will now move to the next part in the SableWasm design, the MIR instructions.

4.3 MIR Instructions

SableWasm MIR uses a control-flow-graph (CFG) based representation in *static single assignment* (SSA) form to represent code body in function definitions. We have provided an introduction to CFG and SSA in the background chapter. Here is a quick recap. CFG splits the control flow within the function into basic blocks. A basic block represents the most extended instruction sequence without control flow transfer, such as branching. Note that for function calls, we take a similar approach to that of LLVM. We will come back to this in detail later in this section. Additionally, SSA requires that all values must have unique definition sites. Hence, in SSA form, the use-definition chain is trivial to compute, while in a traditional CFG, one would need to extract this from the graph with the help of a *reaching definition* analysis. The SableWasm MIR instruction set is similar to WebAssembly bytecode in terms of semantics for most of the instructions. However, it operates over an infinite register machine instead of a stack-based machine, and in some cases, semantics differ in order to keep the size of the SableWasm instruction minimal. In this section, we will cover the design and implementation of SableWasm MIR instructions. The following section will cover the translation strategy between WebAssembly bytecode and the SableWasm MIR and instruction reduction rules. Figure 4.4 provides a general illustration of the design of the SableWasm MIR instruction set. The SableWasm MIR instruction set can currently cover all the instructions defined in WebAssembly specification, including several extensions such as multivalue and SIMD vector operations.

Terminating instructions As discussed above, SableWasm splits the function control flow into basic blocks containing the maximum number of consecutive instructions with-

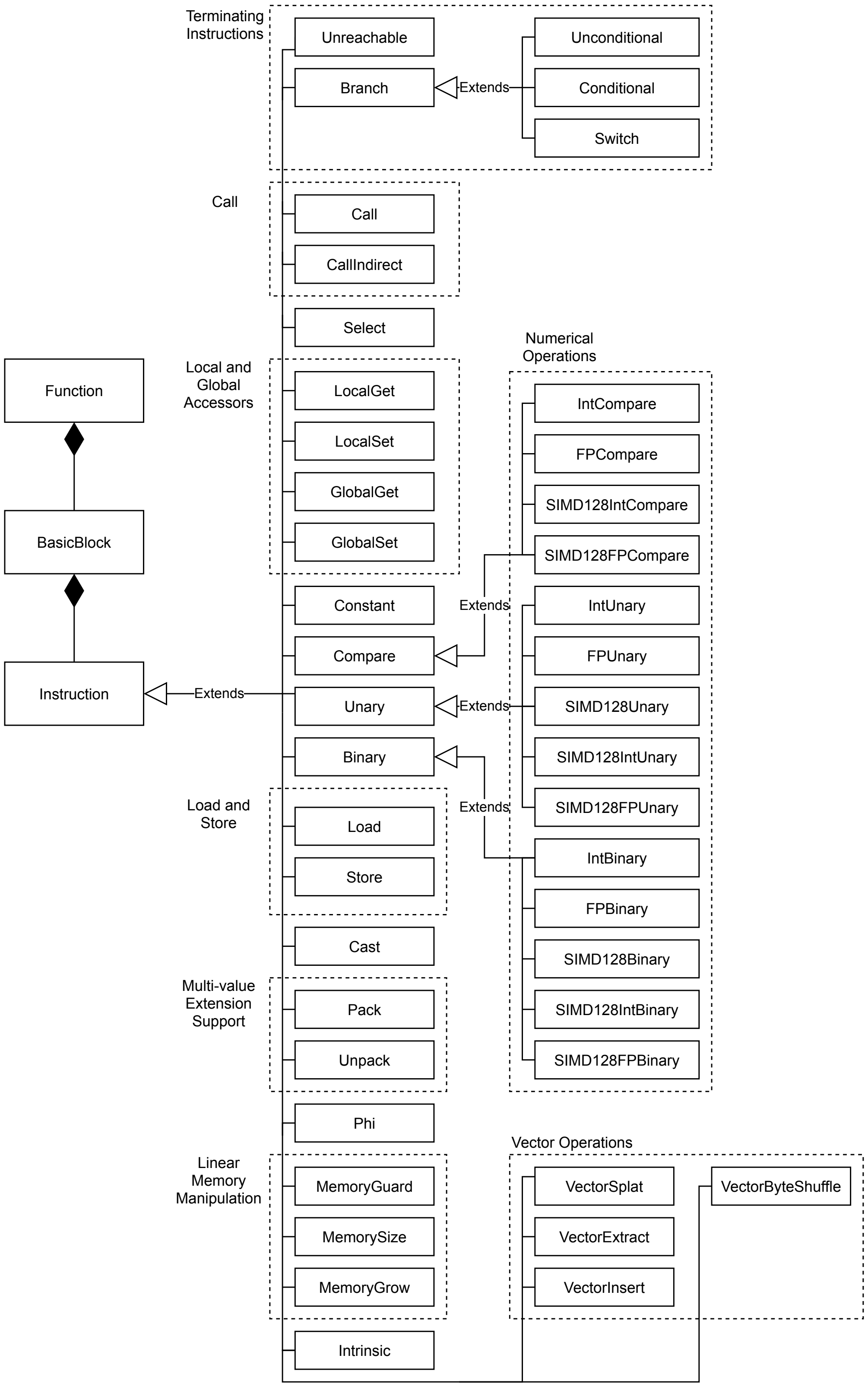


Figure 4.4: SableWasm MIR Instructions

out control flow transfer. In addition, SableWasm, similar to many other SSA form instruction sets, defines a particular group of instructions called terminating instructions. These instructions signal a control flow transfer out of the current basic block, and they must only appear as the last instruction in any given basic block. SableWasm defines four different terminating instructions: unreachable, unconditional branching, conditional branching, and table branching. If the control flow reaches a `Unreachable` instruction, the runtime system will signal a runtime panic. The `Unreachable` instruction in SableWasm is identical to its counterpart in WebAssembly in terms of semantics. The `Unconditional` instruction is an unconditional control flow transfer, as the name suggests. It refers to a target basic block as the operand. At runtime, the instruction will always transfer the control flow to the target basic block. `Unconditional` is similar to the `br` instruction defined in WebAssembly specification. On the other hand, `Conditional` is a conditional branching. It takes a value and two target basic blocks as its operands. At runtime, the instruction will first compare the value against integral value zero. If the value equals zero, the instruction will transfer the control flow to the 'false' basic block, otherwise, to the 'true' basic block. SableWasm's `Conditional` instruction is similar to `br.cond` defined in WebAssembly. The last terminating instruction defined in SableWasm is `Switch`. `Switch` instruction is comparable to the `br.table` instruction in WebAssembly. The instruction takes a value, a list of target basic blocks, and a default branching basic block as its operands. At runtime, `Switch` will interpret the value as an integral value and dispatch accordingly. If the value is within the branching list's range, it will redirect the control flow to the target basic block referred to by the index. Otherwise, `Switch` will transfer the control flow to the default basic block.

Function call In SableWasm, we provide two instructions for function calls defined in WebAssembly specification: direct function calls and indirect function calls. `Call` represents a direct function call where the callee is known at compile time. It takes a function and a list of arguments as operands. On the other hand, `CallIndirect` defines an indirect function call. It implements the indirect function call protocol described in the WebAssembly specification. A quick reminder, in WebAssembly, an indirect function call takes an indirect table, the table index, the expecting function type, and a list of values as arguments. At runtime, the system should first check if the index is valid for the indirect table and fetch the function pointer and its actual signature accordingly. Then, the system should compare the signature against the expecting type. If the signature matches, the runtime system will transfer the control flow to the function referred to by the function pointer. Implementing the signature verification mechanism is backend-specific; we will return to this topic in the next chapter. Note that we do not treat function call instructions as terminating instructions, even though they transfer the control flow to other locations. In SableWasm MIR, we follow the design like that used in the LLVM intermediate representation, where it is assumed that the control flow will continue to the next instruction after returning from the function call. Hence, from the basic block's local perspective, their control flow is pre-determined, and there is no difference compared to other non-terminating instructions.

Local and global variable access In WebAssembly, instructions have access to locals defined by their parent functions and global variables defined by their enclosing module. The SableWasm MIR defines getter and setter instruction for both local and global variables to implement the specification. Their semantics are the same compare to WebAssembly's counterparts. We will skip the detail here, but one can consult the WebAssembly specification for detailed information.

Numerical operations In SableWasm, we classify the numerical operations into three different categories, the `Compare` instructions, `Unary` instructions, and `Binary` instructions. The `Compare` instructions implement the comparison between values, such as 'equal to'. They always yield a 32-bit integer as WebAssembly specification suggests. The `Unary` and `Binary`, as their name suggests, perform unary and binary operations between values. The result of `Unary` and `Binary` instruction is dependent on the opcode.

On the other hand, we can also orthogonally classify the instructions into integer, floating-point, packed integer, and packed floating-point numbers. Note that in MVP WebAssembly, there are only integer and floating-point value operations; the SIMD operation extension proposal adds the packed value operation to the instruction set. In the WebAssembly SIMD extension proposal, the vector value does not store its size and content information in the types. Instead, the packed value instructions' opcodes keep track of the shape of the vector values, which leads to the bloated instruction opcodes. In SableWasm, we separate the instruction opcode from the vector shape. For each of the packed value operations, it must have either a `SIMD128IntLaneInfo` or `SIMD128FPLaneInfo`. Figure 4.4 shows all the classes of numerical operations defined in SableWasm. For detailed opcodes in each numerical instruction class, one can consult SableWasm's source code.

Load and Store `Load` and `Store` instruction provides access to the linear memory for SableWasm MIR. Although in the current version of WebAssembly, the module can contain at most one linear memory, and all WebAssembly's load and store instructions implicitly refer to this linear memory [6]. In SableWasm MIR, we take a different approach. The SableWasm MIR's `Load` instruction takes a linear memory and an integer value as operands. At runtime, the value will be treated as the address (or offset) to the start of the linear memory, and the instruction yields the fetched result. In WebAssembly, the `load` instruction associates with a type and an extension method. For example, `i32.load8_s` loads an 8-bit integer from the linear memory, and then sign extends the fetched byte into a 32-bit integer. In SableWasm, the `Load` instruction associates to a type and an integer value, namely the load width. The load width must equal to or smaller than the width of the type. Also, SableWasm `Load` always perform zero-extension on loaded value. Hence, when translating WebAssembly's sign-extended load into SableWasm's `Load`, one must combine the load instruction with a cast instruction. We will come back to this in chapter 5. The `Store` instruction also associate with a store width. Like the load

[6]This might change in the future version of WebAssembly.

width defined for `Load` instruction, the store width must also be equal to smaller than the store value type's width. The system will first bit truncate the value at runtime and then store the result into the linear memory. One may notice that in SableWasm, we erase the alignment attribute and offset attribute defined in WebAssembly. Currently, we do not support alignment hints from the WebAssembly module. In SableWasm, the `Load` and `Store` always have the alignment requirement of one byte. This implies that the `Load` and `Store` can happen anywhere in the linear memory, corresponding to WebAssembly's linear memory specification.

Linear memory manipulation WebAssembly specification defines two instruction that works with linear memories: `memory.size`, `memory.grow`. Like the WebAssembly's `load` instruction we covered in the previous paragraph, all these instructions operate over the implicitly defined unique linear memory within the module. In SableWasm, we provide similar `MemoryGrow` and `MemorySize` instruction. The semantics of SableWasm's memory manipulation instructions are the same as their WebAssembly counterparts, except that the linear memory needs to be explicitly stated. In SableWasm, we introduce one additional instruction, `MemoryGuard` which is an explicit memory boundary check. In WebAssembly, all `load` and `store` instruction need to check for linear memory out of bound error before access. SableWasm separates the bound check from the memory access. One advantage of this is that one may implement static memory bounds check elimination optimization over SableWasm MIR. Additionally, one backend may provide different strategies for handling boundary checks, such as utilizing invalid virtual memory pages with the operating system's help. In this case, we only need to modify the translation pattern for `MemoryGuard`. `MemoryGuard` takes a linear memory and an integer value as the operand. It also associates with an integer immediate, known as the guard width. At runtime, the system will perform a boundary check over the linear memory starting from the given address to determine if it contains at least a given number of bytes ahead. If there are not enough bytes available, the system should signal a runtime panic.

Pack and Unpack WebAssembly multivalue specification [7] relaxes the constrains on the function type. Functions now can return multiple values instead of at most one value. To support these features, we introduce `Pack` and `Unpack` instructions, along with extending WebAssembly's type system. `Pack` instructions group multiple values into a single ordered tuple, while the `Unpack` reverse the operation by retrieving the value from tuples by index. In the case where a function returns multiple values, we thus use a tuple instead. SableWasm treats tuples as first-class values; however, currently, tuples cannot be recursive. We will come back to this in chapter 5, when we visit the type systems of SableWasm MIR. The index of the `Unpack` must be an immediate value in the current version of SableWasm MIR and is verified at compile time.

Vector operations In the previous paragraph, we introduce the numeric operations defined in SableWasm MIR. However, several instructions do not fit into either `Unary` or `Binary` instructions. Hence, to faithfully support the SIMD operations introduced by the extension proposal, we add four vector-specific operations into SableWasm MIR. They are `VectorSplat`, `VectorExtract`, `VectorInsert` and `VectorByteShuffle`. `VectorSplat` will broadcast the operand value to all lanes in the result vector. SableWasm MIR defines vector splat operation for both packed integer vector and packed floating-point vector. `VectorExtract` is similar to the `extractelement` defined in LLVM intermediate representation. It takes a vector as the operand and also associates itself with an immediate integer value. At runtime, the system extracts the value of the given lane and yields as a result. `VectorInsert` is similar to `insertelement` defined in LLVM. It will replace the vector operand with a given value and yields the updated vector as a result. Note that in the WebAssembly SIMD extension proposal, there are more instructions defined that modify the individual lane value of the vector, such as

[7]WebAssembly Multi-value Proposal: `https://github.com/WebAssembly/multi-value`

`V128Load32Lane` which loads a 32-bit value into a specific lane within the vector. In this project, we would like to keep our instruction set simple; hence, these instructions are reduced into multiple SableWasm MIR instructions. We will come back to this in chapter 5 when we discuss the instruction reduction rules. The last instruction we introduced is the `VectorByteShuffle`. `VectorByteShuffle` is similar to `shufflevector` defined in LLVM, except that it allows rearranging bytes instead of lanes. Currently, the `VectorByteShuffle` only operates over an array of immediate integer values. Compare to the lane shuffle semantics, byte shuffle semantics provides more precise control over the result value. One can trivially simulate a lane shuffle with a byte shuffle. The WebAssembly SIMD extension proposal only defines shuffle for `i8x16`, which corresponding to the byte shuffle semantics. However, in the future, if another shape vector supports shuffle operation, one can generalize the implementation with minimal modification.

Cast `Cast` models the conversion of values to their equivalent form in other types. In SableWasm MIR, we do not distinguish between value conversion and value extension. We treat signed and zero extensions as a kind of value conversion. The `Cast` instruction takes a single value as the operand, and it associates itself with a cast opcode. At runtime, it will perform the conversion according to the opcode, and if the result cannot be accurately represented in the target type, the system should signal a runtime error. The cast opcodes are direct implementations of their WebAssembly counterparts, and we will skip the detail here. One may refer to the WebAssembly specification for more details.

Intrinsic The last SableWasm MIR instruction we are going to cover in this section is the `Intrinsic` instructions. Most WebAssembly instructions can be represented by using the SableWasm MIR instructions, which we covered earlier in this section. However, there are still several corner cases. For example, the WebAssembly SIMD extension proposal defines Q-format rounding multiplication, a type of fix-point multiplication, for packed 16-bit integers. Another example is the `swizzle` operation. A `swizzle` operation is similar to a shuffle operation, except that it takes another vector as the shuffle indices vector instead of an array of immediate integer values. These operations are only defined for a specific vector shape and will introduce unneeded complexity to the SableWasm MIR if we generalize them to all possible vector shapes. Hence, here we group these instructions as the `Intrinsic` instructions. There is no direct mapping to LLVM instruction for most of them, even with the intrinsic functions provided by the framework. Hence, the backend is encouraged to support these instructions with runtime library routines.

In this section, we discussed the design of the SableWasm MIR instruction set, and in the next chapter, we will move to the translation strategy between WebAssembly and SableWasm MIR along with the analysis and transformation framework.

Chapter 5

Middle-level Intermediate Representation Translation and Optimization

The previous chapter presented the SableWasm middle-level intermediate representation (MIR), a static-single-assignment (SSA) control flow graph (CFG) representation of a WebAssembly program. This chapter focuses on the translation strategy used when lowering WebAssembly into the SableWasm MIR. We will first start by presenting the translation patterns used and then discuss the analysis and optimization framework.

5.1 Translating WebAssembly to MIR

In this section, we will cover the translation between WebAssembly bytecode and SableWasm MIR. We have covered the design of SableWasm MIR instructions previously. One may notice that for most of the instructions, especially for the numerical operations, SableWasm MIR shares the same semantics as WebAssembly. Hence, the translation rules for these instructions are pretty trivial, and we will not cover them in detail in this section.

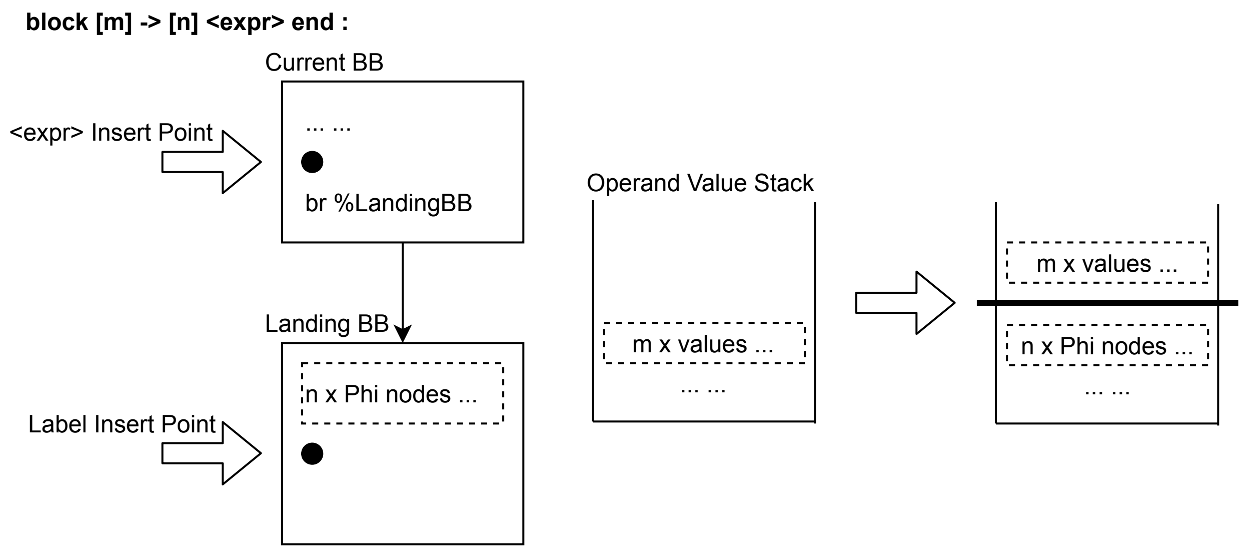


Figure 5.1: WebAssembly `block` translation pattern

Instead, this section will focus on the translation rules for the structured control flow constructs and WebAssembly instructions that require reduction during translation.

5.1.1 Structured-Control-Flow Construct

Translating from stack-based IR to register-based IR is not trivial, especially when non-linear control flow structures appeared. This problem appeared in many runtime system implementations, such as Numba [13], a just-in-time (JIT) compiler for Python. Usually, one needs some algorithm to recover the control flow structure from annoying jump instructions. Luckily, in WebAssembly, we can translate the stack-based bytecode into register-based basic blocks in linear time, thanks to the structured control flow constructs and their validation rules defined in WebAssembly. In this section, we will cover the translation patterns used for WebAssembly's structured-control-flow constructs, namely `block`, `if` and `loop`.

Block In the background chapter, we provide a general illustration of the three structured control flow constructs. As a quick recap, `block` is the simplest form of a structured control flow construct. It implicitly introduces a label at the end of its enclosing instructions. A branch instruction referring to this label will redirect the control flow to the end of the block. Figure 5.1 illustrates the translation pattern for WebAssembly `block` in SableWasm MIR. We will first clarify some of the terminologies we used in the figure, and we will use the same terms later in the `loop` and `if` pattern discussion for consistency. *Expr Insert Point* refer to the starting position for the generated instructions when we recursively translate the instructions within the enclosing expression of the `block` instruction. Furthermore, *Label Insert Point* refer to the position for generated instructions when we finish the recursive translation and resume to the parent expression of the `block` instruction. A *label stack entry* is a tuple consisting of a pointer to the landing basic block, a list of ϕ nodes expecting merge values, and a pointer to the *label insert point*. The translation pattern for `block` is pretty simple; we continue on the current basic block and prepare the landing basic block for the block instruction as a branch instruction within the expression may refer to the label. Additionally, to fully support multi-value extension in WebAssembly, we also need to prepare the ϕ nodes in the landing basic block. SableWasm generates the ϕ nodes based on the type of the `block` instruction. WebAssembly validation rules ensure that the expression within the `block` can access exactly m values from the stack and put n values onto the stack. Finally, we will append an unconditional branch to the landing basic block because in WebAssembly, if the control flow reaches the bottom of the `block` expressions, it will implicitly fall through. For the operand stack, we will first pop m values from the stack as `block` instruction's type suggests and push the ϕ nodes as the result values. Then, we need to set up the boundary between the operand stack for the expression contained within the `block`. Figure 5.1 represents this with the bold line in the result operand value stack. The last step is to push the m values back to the stack, as they are passed to the expression within the `block`.

If The next control-flow structure defined WebAssembly is `if`. WebAssembly's `if` is an expression instead of a statement that appears in many other languages such as C. The `if` expression can yield some values indicated by its type. Figure 5.2 illustrates the

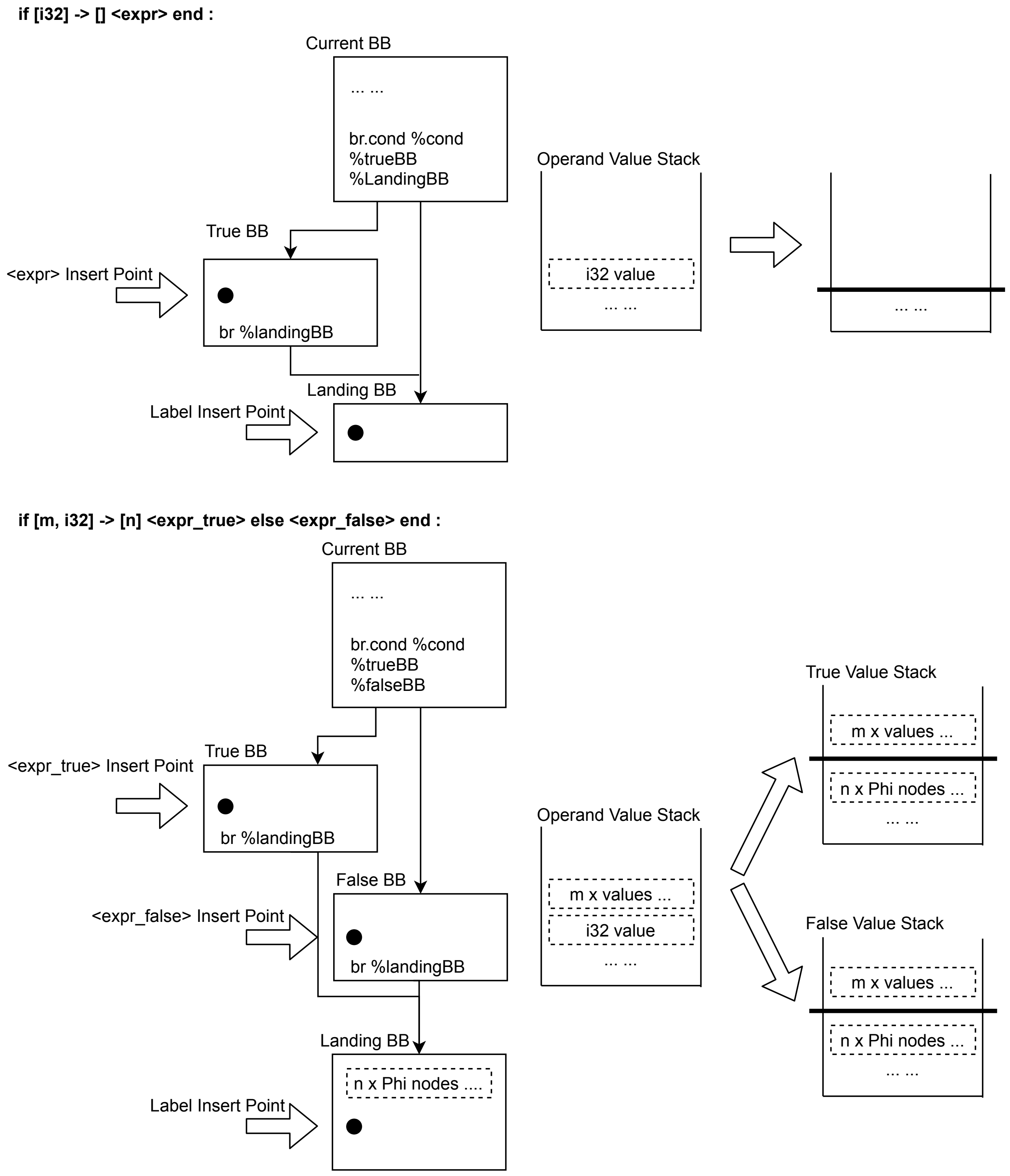


Figure 5.2: WebAssembly `if` translation pattern

translation patterns in SableWasm. There are two types of `if` instruction defined in WebAssembly specification. The first case is a 'partial' `if` instruction, where it only contains the 'true' branch. From WebAssembly validation rules, it's easy to show that the only possible type is `[i32]->[]`, even with the multi-value extension proposal. This implies that the expression within the `if` instruction must start with an empty operand stack. Hence, the translation pattern for the partial `if` is quite straightforward: we only need to pop the condition value from the operand stack and construct a conditional branch based on this value in the current basic block. On the other hand, we also have 'full' `if` instructions with both 'true' and 'false' expressions. The validation rules ensure that both expressions must have the same type. The translation pattern is more complex compare to that of a 'partial' `if`. In this case, we have to prepare the landing basic block similar to what we did for the `block` construct. We need to generate n ϕ nodes for data-flow mergers from the 'true' branch, the 'false' branch, and any possible branch instruction within both nested expressions. Similarly, we need to pop m values from the stack for operand values stack and then push n ϕ nodes. And, within both nested expressions, push m values back to the stack.

Loop The last control-flow structure defined in WebAssembly is `loop`. Figure 5.3 gives a general illustration of SableWasm's translation pattern for `loop` instructions. Similar to the 'partial' `if` we discussed in the previous paragraph, one can show that, under WebAssembly's validation rules, the parameter types for the `loop` instruction must equal to the result types. The `loop` instruction is similar to the `block` instruction, except that if any branch instruction refers to it, the branch instruction should transfer the control flow to the start of the expression within the instruction instead of the end. Thus, we need to prepare a standalone basic block for the nested expression in `loop`, along with the ϕ nodes to merge value on each loop iteration. Note that we also introduce ϕ nodes in the landing basic block. One may argue that there is no need for these ϕ nodes, as only one block can reach the loop exit, and no value merging will occur. Indeed, these ϕ nodes will

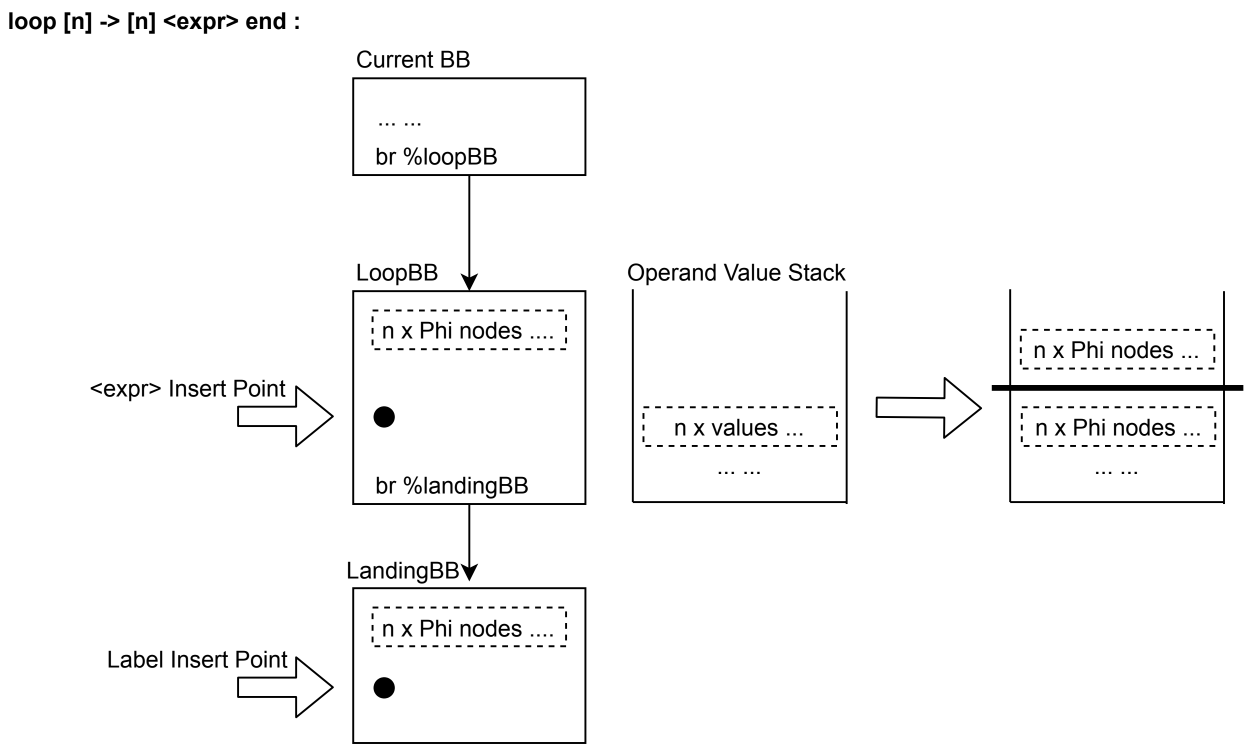


Figure 5.3: WebAssembly `loop` translation pattern

always be trivial ϕ nodes, which have only one possible value inflow. However, this is due to the limitation of our translation framework.

In this section, we discussed the translation patterns for WebAssembly structured control flow constructs. Thanks to WebAssembly validation rules, the types for these structured control flow instructions explicitly mark value merging and imply possible ϕ nodes. Furthermore, one can show that the control graph generated above is indeed in SSA form. However, the directly generated control flow graph is not easily understandable by users. This mainly comes from two facts. First, the WebAssembly-targeting compiler may generate awkward patterns to fit in the structured control-flow constructs. Second, SableWasm translation patterns for structured-control flow constructs are not optimal.

5.1.2 Instruction Reduction

This section will cover the instruction reduction rules used when lowering WebAssembly bytecode to SableWasm MIR. In the background chapter, we mentioned that one of WebAssembly's design goals is to be as compact as possible. Thus, when the community designed the WebAssembly instruction set, they fused several typical instruction sequences into single instructions. For example, SIMD vector operation extension defines `v128.load8x8_s` which first load 8 8-bit integers into a vector, and then sign-extends them into 16-bit integers. Another example will be `v128.load32_lane` which loads a 32-bit value, either a 32-bit integer or a single-precision floating-point number into a given vector. Such design is understandable for WebAssembly as binary size does matter when shipping applications over the internet. But, for SableWasm, a static compiler, we focus more on the size of the instruction set instead of the size of the intermediate representation. It is harder to write analysis for a bloated instruction set, as one needs to consider more instruction cases. Hence, when lowering WebAssembly bytecode to SableWasm MIR, we replace some WebAssembly instructions with SableWasm MIR instructions sequences.

Eqz

```
[..., %n i32] i32.eqz ⟹ %t0 = i32.const 0; %t1 = int.eq %n %t0
[..., %n i64] i64.eqz ⟹ %t0 = i64.const 0; %t1 = int.eq %n %t0
```

WebAssembly defines a unary `eqz` operations for all integer values. As the name suggests, `eqz` compares the operand value against zero and yields one if true, zero otherwise. In SableWasm MIR, we group all comparison instructions into the `Compare` class, and `eqz` does not fit into the class as it is not a binary operation. Hence we rewrite the `eqz` as `Compare` instruction with opcode as `Eq`.

Load

```
[..., %base i32] i32.load offset=%offset align=%align ⟹
    %addr = int.add %base %offset
    memory.guard %mem %addr 4
    %t0 = load.32 i32 %mem %addr
[..., %base i32] i32.load16_s offset=%offset align=%align ⟹
    %addr = int.add %base %offset
    memory.guard %mem %addr 2
    %t0 = load.16 i32 %mem %addr
    %t1 = cast i32.extend.16.s %t0
[..., %base i32] i32.load16_u offset=%offset align=%align ⟹
    %addr = int.add %base %offset
    memory.guard %mem %addr 2
    %t0 = load.16 i32 %mem %addr
```

In the SableWasm instruction design section, we introduced the `Load` and `MemoryGuard` in SableWasm MIR. A quick recap, SableWasm MIR `Load` instruction, compare to its WebAssembly counterpart, assumes access is in-bound, does not support offset attribute, and always performs zero-extension on partial loads. Hence, to properly support WebAssembly's `load` instructions, we need to reduce them with the strategy shown above. For load instructions that do not require value extensions, such as `i32.load`, we first calculate the actual starting address, perform a memory boundary check with `MemoryGuard`, and then perform the memory read. On the other hand, for a partial load operation, we need first to perform the load operation using the same protocol as a normal load. Then, if a sign extension is needed, we will add its corresponding cast instruction. In the example above, we demonstrate this with WebAssembly's `i32.load16_s`. In this case, SableWasm appends a `Cast` instruction with opcode `i32.extend.16` after the load operation.

Store

```
[..., %base i32, %val i64] i64.store offset=%offset align=%align ⟹
    %addr = int.add %base %offset
    memory.guard %mem %addr 8
    store.64 %mem %addr %val
[..., %base i32, %val i64] i64.store16 offset=%offset align=%align ⟹
    %addr = int.add %base %offset
    memory.guard %mem %addr 2
    store.16 %mem %addr %val
```

Similar to the `Load` instruction we discussed earlier, the `Store` instruction also assumes the memory access is always in range and does not provide the offset attribute. However, a `Store` instruction will always perform truncation instead of extension. Further, the only possible truncation is the bit-truncation by discarding bits starting from the most significant bit. The instruction reduction rules for WebAssembly `store` instructions is similar to those for `load` instructions. In the example above, we demonstrate the rules with `i64.store` and its partial store version, `i64.store16` which only stores the lowest two bytes into linear memory. SableWasm inserts `MemoryGuard` instructions in a similar fashion to `load` instructions. Note that we do not insert an explicit `Cast` instruction to perform the truncation. A `Store` instruction will implicitly truncate the value according to the store width; in this case, it will truncate the 64-bit integer into a 16-bit integer.

SIMD extension proposal reduction rules

```
[..., %lhs v128, %rhs v128] v128.andnot ⟹
    %t0 = v128.not %rhs
    %t1 = v128.and %lhs %t0
[..., %lhs v128, %rhs v128] i16x8.extmul_low_i8x16_s ⟹
    %t0 = cast i16x8.extend.low.i8x16.s %lhs
    %t1 = cast i16x8.extend.low.i8x16.s %rhs
    %t2 = v128.int.mul i16x8 %t0 %t1
[..., %lhs v128, %rhs v128] i16x8.extmul_low_i8x16_u ⟹
    %t0 = cast i16x8.extend.low.i8x16.u %lhs
    %t1 = cast i16x8.extend.low.i8x16.u %rhs
    %t2 = v128.int.mul i16x8 %t0 %t1
```

The SIMD extension proposal introduces approximately 240 instructions into the WebAssembly instruction set. However, not all of them are simple single operation instructions. The SIMD extension proposal also follows WebAssembly's design goal to ensure the compactness of the generated program. The proposal suggests reduction rules for several SIMD operation instructions, and in SableWasm, we take advantage of them to reduce the size of the instruction set. The first applicable instruction is the `andnot` operation for vectors. The `andnot` is equivalent to performing bitwise 'not' on the right-hand-side operand, and then a bitwise 'and' operation between the left-hand-side operand and the temporary result. SableWasm reduces `andnot` into a `not` instruction followed by a and instruction, as shown in the example above. The second group of reducible instructions is the ExtMul instructions. The SIMD extension proposal defines ExtMul for all packed integer vectors except packed 64-bit integers. They are equivalent to first widening the vector using the appropriate extension and then multiplying two operands. In the example above, we demonstrate with i16x8.extmul_low_i8x16_s which performs an ExtMul operation for packed 8-bit integers. SableWasm implements this instruction by first performing a sign extension on the lower half of the vector and multiplying the temporary result as shown above. SableWasm also applies a similar procedure to i16x8.extmul_low_i8x16_u, except that it uses a zero-extension in the Cast instruction instead of sign-extension.

SIMD load with zero-padding

```
[..., %base i32] v128.load32_zero offset=%offset align=%align ⟹
    %addr = int.add %base %offset
    memory.guard %mem %addr 4
    %t1 = load.32 i32 %mem %addr
    %t2 = const v128 0
    %t3 = v128.int.insert i32x4 0 %t2 %t1
```

The WebAssembly SIMD extension proposal also introduces many variations of load operations. The first variation is the 'zero-padding' load operation. The 'zero-padding' load is equivalent to loading a scalar from the linear memory and then inserting it into a zero-initialized vector. We demonstrate this with the example above. We first use the protocol we discussed above to load a scalar 32-bit integer. Then, we insert it into a zero vector using VectorInsert instruction. The WebAssembly SIMD extension proposal defines 'zero-padding' load operations for all packed integers and packed float-point numbers. The reduction rules for them are similar to the pattern above.

SIMD load and splat

```
[..., %base i32] v128.load32_splat offset=%offset align=%align ⟹
    %addr = int.add %base %offset
    memory.guard %mem %addr 4
    %t1 = load.32 i32 %mem %addr
    %t2 = v128.int.splat i32x4 0 %t1
```

The second variation of SIMD vector load is the 'load-and-splat' load operation. This type of load operation is a combination of scalar load operation and vector splat operation. It first loads a scalar from the linear memory and then broadcasts the value to all vector lanes. SableWasm uses a similar reduce rule compared to the 'zero-padding' load operation, except that instead of inserting the scalar into a zero-initialized vector, we use `VectorSplat` to broadcast it. The example above demonstrate this with `v128.load32_splat`. Similar to the 'zero-padding' load operation, 'load-and-splat' is defined for all packed integers and packed float-point numbers.

SIMD load lane

```
[..., %base i32, %vec v128]
v128.load32_lane offset=%offset align=%align lane=%lane ⟹
    %addr = int.add %base %offset
    memory.guard %mem %addr 4
    %t1 = load.32 i32 %mem %addr
    %t2 = v128.int.insert i32x4 %lane %base %t1
```

The next variation of the SIMD vector load operation is the 'load-lane' load operation. The example above demonstrates the procedure with a sample of WebAssembly's `v128.load32_lane` which reads a 32-bit integer from linear memory and inserts it into a specific lane of a given vector. SableWasm first lowers the load semantic using the same protocol as we discussed above and then inserts to the given vector using the `VectorInsert` instruction. Again, the WebAssembly SIMD extension proposal defines 'load-lane' load operation for all shapes of packed integers and floating-point numbers. In WebAssembly SIMD load operation variations, one may already notice that we only have a width associated with them instead of types. This is because WebAssembly SIMD operations do not distinguish the shape of the vector. Hence, there is no difference in loading a 32-bit integer and a single-precision floating number, as they both consume 32-bit storage. But in SableWasm, we distinguish between packed integers and packed floating-point numbers for the SIMD instruction shape record. On the other hand, SableWasm also erases shape information from the vector value, and it is the responsibility of the instruction to interpret the value correctly. Thus, when we perform a load operation, we always assume that we are loading packed integers. In the examples above, the 32-bit load with translate to 'load a 32-bit integer'.

SIMD load and extend

```
[..., %base i32] v128.load16x4_s offset=%offset align=%align ⟹
    %addr = int.add %base %offset
    memory.guard %mem %addr 8
    %t1 = load.64 v128 %addr 8
    %t2 = cast i32x4.extend.low.i16x8.s %t1
[..., %base i32] v128.load16x4_u offset=%offset align=%align ⟹
    %addr = int.add %base %offset
    memory.guard %mem %addr 8
    %t1 = load.64 v128 %addr 8
    %t2 = cast i32x4.extend.low.i16x8.u %t1
```

The last variation of a load operation is the 'load-and-extend' load operation. It is a combination of partial load and extension on the lower half of 128-bit vectors. In the example above we present examples for `v128.load16x4_s` and `v128.load16x4_u`. The previous instruction loads four 16-bit integers into the lower lanes of the vector and performs sign-extension on the result to get a packed 32-bit integer vector. `v128.load16x4_u` performs a similar operation, except that it performs zero-extension instead of sign-extension. A quick reminder, SableWasm MIR `Load` instruction can apply to any primitive value type and supports partial loading by annotating with a smaller load-width. In the case of the partial load, SableWasm MIR `Load` always loads bytes starting from the least significant bit and performs zero-extension on the result. SableWasm takes advantage of the `Load` instruction's design when lowering the 'load-and-extend' load operation. In the example above, we partially load a 128-bit vector with a 64-bit value which corresponds to loading four 16-bit integers from the linear memory. Note that this `Load` instruction yields a vector of 16-bit integers with four zero values in its higher lanes and loaded values in its lower lanes. Thus, we only need to perform a `Cast` operation with opcode `i32x4.extend.low.i16x8.s` to reach the desired result. SableWasm treats `v128.load16x4_u` using a similar procedure, except that it uses zero-extension instead of sign-extension. Finally, like other load operation variations discussed above, WebAssembly defines the 'load-and-extend' load operation for all packed integer and packed floating-point numbers.

SIMD store lane

```
[..., %base i32, %val v128]
v128.store32_lane offset=%offset align=%align lane=%lane ⟹
    %addr = int.add %base %offset
    memory.guard %mem %addr 4
    %t1 = v128.int.extract i32x4 %val %lane
    store.32 %mem %addr %t1
```

Similar to the 'load-lane' load operation variation, the WebAssembly SIMD extension proposal also defines direct lane store instruction for 128-bit vectors. The above example demonstrates the reduced rules for these instructions. Let's take `v128.store32_lane` as example. SableWasm MIR first calculates the address and sets up a memory boundary check use a protocol similar to what we have seen above. Then, it extracts the lane value by using `VectorExtract` instruction and stores it into linear memory. Like WebAssembly `load` instructions, the `store` instruction does not distinguish between packed integers from packed floating-point numbers. In SableWasm, we always assume the store vector is packed integers.

5.2 Analysis Framework

SableWasm also implements an analysis and optimization framework over its middle-level intermediate representation (MIR). The framework consists of two parts, passes and drivers. The SableWasm analysis and transformation framework only provides essential support for managing passes, compared to other more advanced frameworks, such as McSAF [3], an optimization framework for MATLAB language. Figure 5.4 illustrates the current state of the framework in SableWasm. Currently, we implement three different drivers. `SimpleModulePassDriver` accepts module passes and operates on the module level. At the time of thesis writing, we haven't explored inter-

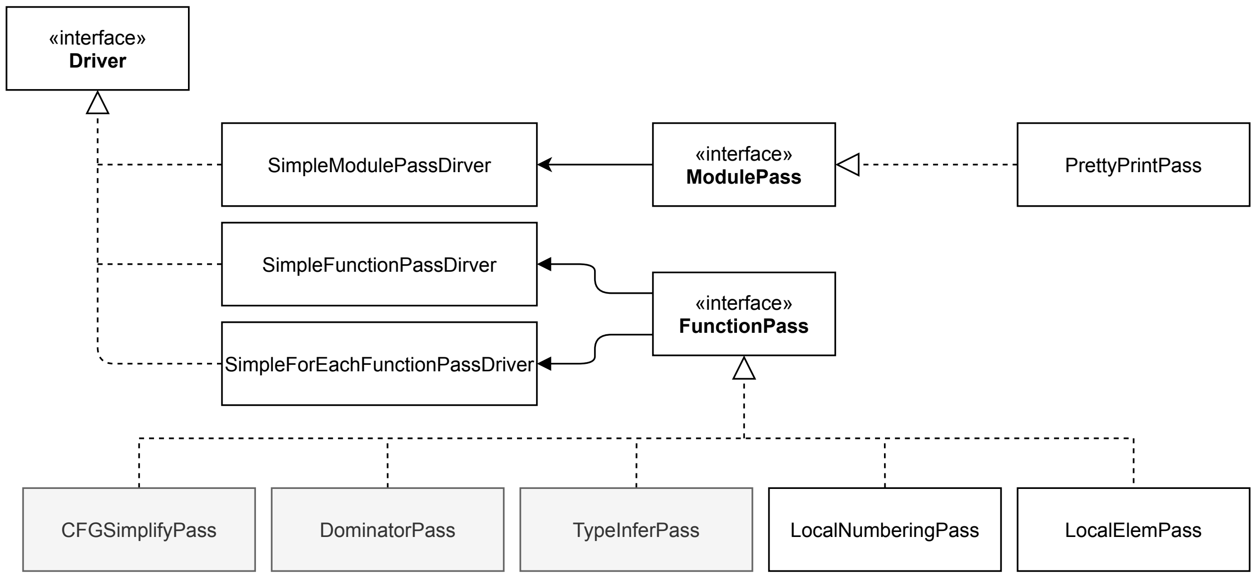


Figure 5.4: SableWasm MIR Analysis and Optimization Framework

procedural analysis for SableWasm MIR in detail, and the only module pass implemented is the pretty-print pass. In the future, one can add additional inter-procedural analyses to SableWasm, by implementing the `ModulePass` interface. The second driver is the `SimpleFunctionPassDriver`. As its name suggests, it manages `FunctionPass` instead. `FunctionPass` implements intra-procedural analysis that operates over basic blocks. SableWasm currently implements multiple intra-procedural analyses, such as dominator tree construction. We will cover these passes in detail in this section. The last driver in SableWasm is `SimpleForEachFunctionPassDriver` which is a wrapper class for `SimpleFunctionPassDriver`. It works with `FunctionPass` but takes a module as an argument.

5.2.1 Dominators and Dependence

Dominator tree and immediate dominance are close related to *static single assignment* (SSA) form, and Ron Cytron's classic paper on converting control flow graph (CFG) to SSA [2] shows that SSA directly derives from them. The dominator tree represents the dominance relationship between basic blocks. A basic block is a *dominator* of another if all control flow reaching the later block must go through the first block. On the other hand, *immediate dominance* defines a stricter relationship between basic blocks. A basic block is an immediate dominator of another if it satisfies two conditions. First, the candidate block must be a dominator block of the second one. Second, it does not dominate any other blocks that dominate the second block. Although the SableWasm MIR is already in SSA form, the dominator tree is still helpful in the later analysis and backend code generation. One may notice that the dominator relationship in SSA is comparable to the same problem in graph theory. Indeed, they are the same problem if we treat the basic blocks as vertices and control flow paths as edges among them. A direct solution to compute the dominator set utilizes forward analysis within $O(n^2)$, respecting to the number of basic blocks in the CFG. More efficient algorithms can yield the dominator set within almost linear time, such as Tarjan's algorithm [17], and its refined version [6]. Currently, SableWasm compiles programs usually with smaller functions that contain approximately 200 basic blocks at most. Hence, an efficient complex algorithm does not have too much room for improvement. In the future, if this becomes the bottleneck of the compilation pipeline, one should replace the implementation with a better algorithm. This section will present the forward analysis implementation briefly, and it is a classic implementation for dominator tree construction.

Formalisms In the rest of the section, we will use $dom(\cdot)$ to represent the set of *strict dominators* for a given basic block. The set of strict dominators for A is the set of dominators for A subtracting A itself. Hence, 'block A is a strict dominator for block B' implies that $A \in dom(B)$. Similarly, BB_{idom} is an immediate dominator for basic block A, if and only if, $BB_{idom} \in dom(A)) \wedge (\forall B \in dom(A), BB_{idom} \notin dom(B)$. Finally, the dominator tree represents all basic blocks with tree nodes and adds directed edges according to the immediate dominator relationship.

Dataflow analysis The algorithm is a classic forward dataflow analysis. In this paragraph, we will quickly cover the key points in the algorithm. For more detailed information, one should consult Cytron's paper on SSA construction. During pass initialization, we first set the following, $\forall A \in BB \setminus \{BB_{entry}\}, dom(A) = BB$, where BB denotes the set of all basic blocks that appeared in the control flow graph, and BB_{entry} denotes the entry basic block for CFG. For the entry basic block, we set $dom(BB_{entry}) = \{BB_{entry}\}$ instead. The initialization value is a conservative guess of the result, and the next step is to refine it. The iterative step rule is as follow,

$$\forall A \in BB, dom(A) = \left\{ \{A\} \cup \bigcap_{B \in pred(A)} dom(B) \right\}$$

Here, $pred(\cdot)$ denotes the predecessor of the given basic block. The general idea is that a basic block that dominates all its predecessors must also dominate the given basic block for each of the basic blocks. The stop criteria for the dominator analysis are also quite simple. If there are no more changes in the result, the forward analysis will terminate.

Implementation SableWasm implements the forward dataflow analysis we discussed above with class `DominatorPass`. In addition, the analysis pass object shares its result with a helper class `DominatorPassResult` which provides helper methods for accessing the result, such as calculating the immediate dominator and constructing the dominator tree from the result sets. Finally, SableWasm uses several techniques to improve the performance, such as modeling the set with sorted arrays.

In this section, we presented the dominator analysis in SableWasm. The dominator analysis is quite common among compiler implementations, and it will play a critical role in the later part of the project.

5.2.2 Control-Flow Graph Simplification

In section 5.1, we illustrated the translation rules from WebAssembly bytecode to SableWasm MIR. Unfortunately, the translation rules yield suboptimal control flow graphs. Hence, in this section, we will incrementally improve the control flow graphs by fixing

```
function %foo : [i32] -> [i32] {
{%operand:i32}
%entry:  #pred = {}
  %0 = local.get %operand
  %1 = const i32 2
  %2 = int.rem.s %0 %1
  %3 = int.eqz %2
  br.cond %3 %BB:0 %BB:1

%BB:0:   #pred = {%entry}
  br %BB:3

%BB:1:   #pred = {%entry}
  br %BB:4

%BB:2:   #pred = {}
  br %BB:3

%BB:3:   #pred = {%BB:2, %BB:0}
  %4 = const i32 1
  br %exit

%BB:4:   #pred = {%BB:1}
  %5 = const i32 0
  br %exit

%exit:   #pred = {%BB:4, %BB:3}
  %6 = phi i32 [%4, %BB:3] [%5, %BB:4]
  ret %6
}
```

```
(func $foo
  (param $operand i32) (result i32)
  block  ;; label = @1
    block  ;; label = @2
      local.get $operand
      i32.const 2
      i32.rem_s
      i32.eqz
      if  ;; label = @3
        br 1 (;@2;)
      else
        br 2 (;@1;)
      end
    end
    i32.const 1
    return
  end
  i32.const 0)
```

Figure 5.5: Control-flow graph simplification example

several obvious issues we found, such as trivial ϕ nodes and unnecessary branching. The control flow graph simplification also performs *dead code elimination* and *unreachable basic block elimination*. This section presents the patterns, along with their transforming strategies used in SableWasm. The general design of the simplification pass is similar to what one would expect in a peephole optimizer [19]. It iterates through the control flow graph, scans for matched patterns, and if it finds any optimization opportunities it will apply transformation strategies immediately. In the future, one may generalize this simplification pass into a fully-featured peephole optimizer, using a domain-specific language for patterns similar to Alive [16, 18] for LLVM to ensure extensibility and correctness of the patterns. The simplification pass will terminate once the execution reaches a fixed point, where there are no more optimization opportunities.

Trivial ϕ nodes The first pattern we found in generated SableWasm MIR is the trivial ϕ nodes. Trivial ϕ nodes refer to the ϕ nodes with only one candidate value. In section 5.1.1, we present the translation patterns for `loop` instructions in WebAssembly and mentioned that the pattern is suboptimal and will result in trivial ϕ nodes. A quick reminder, the `loop` instruction needs to insert ϕ nodes to the landing basic block, which necessarily has non-merging control flow as an effect of a limitation in our translation framework. To address this, we search for `%t0 = phi t [%t1, %path]` for all possible type t. The transformation strategy is to replace all appearances of value `%t0` with value `%t1`. As the ϕ nodes do not map to any operations and are only introduced by SSA to explicitly mark value merging, removing them from the control flow graph does not change the semantics of the program. When replacing the values, SableWasm uses the use-site lists managed by the `ASTNode` to boost the performance.

Redundant branching The second pattern focus on redundant branching. Redundant branching can also come from the translation patterns for structured control flow. One may already notice that we will always generate a landing basic block for the instruction for every structured control flow construct. However, when the control flow constructs are the last instructions in their enclosing expression, the landing basic blocks will only contain a single branching instruction. Figure 5.5 demonstrates an unoptimized example. On the right-hand side, the WebAssembly function is a simple function that returns one when the operand is an even number and zero otherwise. On the left-hand side is its corresponding SableWasm MIR before simplification. Clearly, `%BB:0` and `%BB:1` are redundant. The redundant branch elimination pattern looks for basic blocks with a single inward flow and attempts to merge them with their predecessors. In the example, the optimizer will try to merge `%BB:1` and `%BB:4` by moving the `Constant` instruction into `%BB:1`, and redirecting the branching in `%BB:1` from `%BB:4` to `%exit`.

Dead basic block The third pattern we have in SableWasm to simplify the control flow graph is dead basic block elimination. In figure 5.5, we have a dead basic block, namely

%BB:2. These dead basic blocks again come from SableWasm's translation patterns. When we are translating the control flow constructs, we always prepare the landing basic block. However, in many cases, the control flow may not reach the landing basic block. In the example above, we have a WebAssembly `return` instruction appear in the `block`'s nested expression. The translation patterns for `return` instruction is naive, which creates a branch to the exiting block and configures the ϕ nodes accordingly. Hence, in this case, the landing basic block will never have an inward flow. In SableWasm MIR, we do not consider these unreachable basic blocks malformed. However, in many backends, these are considered bad behaviour. In addition, these basic blocks also interfere with other optimizations. In the example in figure 5.5, `%BB:3` does not satisfy the redundant branching elimination pattern because it does not have a unique inward flow. However, one of them, `%BB:2`, is a dead block. Thus, we may find more optimization opportunities by removing dead basic blocks from the control flow graph. In SableWasm, we identify the dead basic block via a mark-and-sweep algorithm. Starting from the entry block, we mark all the basic blocks that are reachable. Then we iterate overall basic blocks, and if the basic block does not have the flag, we add them to the delete list. Finally, we remove all the basic blocks within the delete list from the control flow graph.

Dead value The last pattern we have in the control flow graph simplification pass is dead value elimination. Dead value elimination is similar to the dead basic block elimination, except that it works with values instead of basic blocks. Unfortunately, the example in figure 5.5 does not contain any dead values. However, the idea is quite simple to understand. Most of the dead values come from WebAssembly's `drop` instruction which discards values from the implicit operand stack. In a non-SSA control flow graph, one usually needs first to perform *liveness analysis* and *reaching definition analysis* to determine if the value is dead. But in SSA, one can quickly recover this information from the use-definition chain, and in SableWasm, the base class `ASTNode` automatically manages it. Thus, the optimizer will iterate over all values within the control flow graph and check

```
function %foo : [i32] -> [i32] {
{%operand: i32}
%entry:  #pred = {}
  %0 = local.get %operand
  %1 = const i32 2
  %2 = int.rem.s %0 %1
  %3 = int.eqz %2
  br.cond %3 %BB:0 %BB:1

%BB:0:   #pred = {%entry}
  %4 = const i32 1
  br %exit

%BB:1:   #pred = {%entry}
  %5 = const i32 0
  br %exit

%exit:   #pred = {%BB:1, %BB:0}
  %6 = phi i32 [%4, %BB:0] [%5, %BB:1]
  ret %6
}
```

Figure 5.6: Control-flow graph simplification result

if others refer to it. If not, it then verifies if the instruction is *droppable*. A droppable instruction is an instruction such that if we remove it from the control flow graphs, no observable effects should happen, similar to the concept of 'pure' for functions. Finally, if instructions are both dead and droppable, the optimizer will remove them from the control flow graph.

In this section, we covered the flow graph simplification pass in SableWasm. The optimizer will iteratively run four patterns that we have discussed above until it reaches a fixed point. Figure 5.6 shows the result of running these optimizations on the input shown in figure 5.5. Compared to the original, the simplified version is more readable. Moreover, by reducing the number of basic blocks, we can improve other analyses in SableWasm.

5.2.3 Type Inference

This section presents the type system for SableWasm MIR. SableWasm MIR is a statically typed language with a pretty straightforward type system. However, one may already notice that SableWasm MIR does not annotate every instruction with a type, unlike many other compiler intermediate representations. Instead, SableWasm computes the types for values on-demand via a set of type inference rules. The type system for SableWasm MIR generalizes from the MVP WebAssembly type system and its extension proposals with a few modifications. The formal definition for SableWasm MIR types are as follow,

```
⟨primitive_type⟩ ::= i32 | i64 | f32 | f64 | v128
⟨tuple_type⟩     ::= (N, ⟨primitive_type⟩...)
⟨type⟩           ::= ⟨primitive_type⟩ | ⟨tuple_type⟩ | () | ⊥
```

Here we will skip the discussion for *primitive type* and the type checking rules for its corresponding instructions as they are equivalent to the MVP WebAssembly type system. The *tuple type* consists of an unsigned integer and a list of primitive types. They model the return types of multi-value return functions or `Pack` instructions. Finally, we introduce the unit type, (), and the bottom type, $\perp$. One can consider the unit type as `void` in the C programming language. It represents no value present, but the type is valid. On the other hand, the bottom type, $\perp$, signals that the pass cannot assign any valid type to the term. In the rest of this section, we will focus our discussion on extensions made due to two major WebAssembly extension proposals, multi-value and SIMD operation.

Multi-value WebAssembly multi-value extensions allow functions to have more than one return values, which is quite interesting. Usually, low-level bytecode representation does not directly support this feature, and it usually only appears in higher-level language designs, such as Python. In section 4.3, we introduced two instructions `Pack` and `Unpack`, along with how we represent multi-value for functions. As a quick recap, SableWasm uses tuples to denote the multi-value return for functions. The `Pack` instruction collects values and constructs a tuple containing them, while on the other hand, the `Unpack` extracts primitive values from tuples. Let's focus on the `Pack` instruction first. The typing rule for `Pack` is straightforward. If we can infer types for all candidate values, we say that the `Unpack` instruction has a tuple type consisting of the number of candidate values and a list of element types. On the other hand, if any of the candidate values result in a non-primitive type, the `Pack` instruction is the $\bot$ type. More formally,

$$\frac{\Gamma \vdash v_0 \Rightarrow t_0, \ldots, v_n \Rightarrow t_n \qquad \forall i, t_i \in \mathit{primitives}}{\Gamma \vdash \textbf{pack}\, v_0, \ldots, v_n \Rightarrow \langle n, t_0 \ldots t_n \rangle} \qquad \frac{\Gamma \vdash \exists i, v_t \notin \mathit{primitives}}{\Gamma \vdash \textbf{pack}\, v_0, \ldots, v_n \Rightarrow \bot}$$

Here the set $\mathit{primitives}$ is the set of all possible primitive types in the SableWasm MIR type system. For `Unpack` instructions, the type checker will first check if the immediate index is within the tuple size. If the index is out of bounds, the type checker will assign the instruction with bottom type $\bot$. Otherwise, it will take the type from the tuple specified by the index. Formally,

$$\frac{\Gamma \vdash v \Rightarrow \langle n, t_0 \ldots t_n \rangle \qquad 0 \leq k \leq n}{\Gamma \vdash \textbf{unpack}\ \ \mathrm{k\, v} \Rightarrow t_k} \qquad \frac{\Gamma \vdash v \Rightarrow \langle n, t_0 \ldots t_n \rangle \qquad \text{otherwise}}{\Gamma \vdash \textbf{unpack}\ \ \mathrm{k\, v} \Rightarrow \bot}$$

We also generalize the function type in WebAssembly so that SableWasm MIR's function type will always have a single return value. We use the following strategy to map WebAssembly's function type into SableWasm MIR function type. In the case where there are no return values, we translate the return type into unit type. For example, SableWasm translate `[i32] -> []` into `[i32] -> ()`. On the other hand, if the function type has exactly one return value, the translation rule is trivial. Finally, when there are multiple return values, we pack them into a single tuple. For example, SableWasm use `[i32] -> (2, i32, f32)` to represent `[i32] -> [i32, f32]` in WebAssembly.

SIMD operations Section 4.3 presented the instruction design in SableWasm MIR. We mentioned that WebAssembly's 128-bit vector value, added by the SIMD operation extension proposal, does not store their shape information in the type. WebAssembly's design gives us two choices in SableWasm when designing a type system for vector operations. First, we can erase all the shape information for values and carefully plan the instruction semantics to ensure that all the operations have defined behaviour at runtime. Second, another approach is to add shape information back to the values' types. If there is a mismatch in shape information, either the translation visitor can insert a bit cast, or the type checker can reject the program. In SableWasm MIR, we take the first approach by erasing all the shape information from the vector values. Chapter 6 will introduce the second approach in detail. The semantics for SIMD instructions in SableWasm MIR follows the WebAssembly's specification. We always store the value using the little-endian method and the vectors start their first lane from the least significant bit.

In this section, we talked about the type inference pass in SableWasm MIR. Similar to the dominator analysis we seen in section 5.2.1, the type infer pass does not optimize the control flow graph. But they are critical in the backend when we lower the SableWasm MIR into LLVM. We will come back to this in detail in chapter 6.

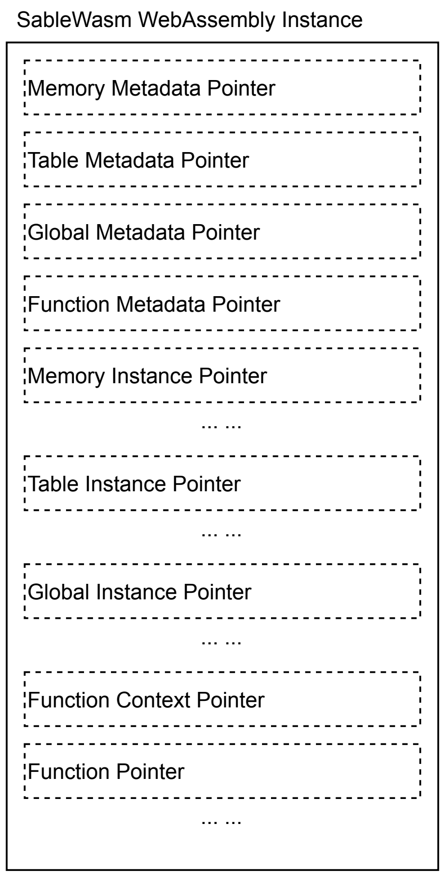
Chapter 6

Backend and Runtime

This chapter discusses the last component of the SableWasm compilation pipeline: the code generation backend and runtime support for generated shared libraries. Currently, SableWasm has only one backend based on the LLVM compiler infrastructure. However, in the future, one can easily extend the system by adding more backends that lower SableWasm MIR into other target languages. Another problem that appears when designing a backend is how SableWasm MIR entities map to native constructs. In SableWasm, we take an instance-based approach. The SableWasm runtime library will manage all entities in an instance object. The system will pass it to the generated native functions as the first argument, similar to 'this' pointer in many C++ implementations. In the rest of this chapter, we will first go through the design of the instance object, followed by the implementation of WebAssembly entities. Finally, we will discuss the code generation strategies used when lowering SableWasm MIR to LLVM intermediate representation and the interaction between generated shared libraries and the hosting language.

6.1 Instance Layout

This section discusses the WebAssembly instance implementation in SableWasm. A WebAssembly instance hosts all the runtime structures that the generated shared libraries



```
struct instance {
    memory_metadata_t   *memory_metadata;
    table_metadata_t    *table_metadata;
    global_metadata_t   *global_metadata;
    function_metadata_t *function_metadata;
    memory_t            *memories[NUM_MEMORY];
    table_t             *tables[NUM_TABLE];
    global_t            *globals[NUM_GLOBAL];
    struct {
        struct instance *context;
        function_t      *function_ptr;
    } *functions[NUM_FUNCTIONS];
};
```

Figure 6.1: SableWasm WebAssembly instance

require, such as linear memories and indirect tables. Figure 6.1 illustrates the design of the WebAssembly instance. SableWasm's WebAssembly instance object consists of two parts, metadata entries and entity pointers. One may also notice that the instance object's size may vary from one module to another depending on how many entities are declared. This behaviour is intentional by design. The SableWasm runtime system needs to compute the address of the pointers based on the metadata information on the fly. By packing all pointers in a consecutive memory region, we reduce one layer of indirection for the runtime system, and in theory, may improve runtime performance. On the other hand, the generated shared library has all the entities address inlined as the backend can compute them during code generation, which does not incur any performance loss. For most of the entities, they are pretty straightforward, and we will skip the discussion here. In the rest of the section, we focus on three aspects: the metadata entries, the function entity representations, and the instance initialization protocol in SableWasm.

Metadata One could think of the metadata as the signatures for entities, and indeed, the SableWasm runtime system prepares the instance object based on the metadata. Further, shared libraries generated by SableWasm only publicly expose the metadata and initialization function to conceal module details. Metadata encodes the type for the entity. For linear memories and indirect tables, this is relatively trivial as their types only consist of an integer pair. In the case of global variables, things are a little bit complicated. A quick reminder, WebAssembly global variable types keep track of their value type and mutability. The first problem here is how to encode WebAssembly value types. One solution is to use WebAssembly value type binary format. However, this encoding strategy is hard to maintain as a human cannot directly read them. Here we use the JVM approach for value type encoding [1]. In short, in SableWasm, we encode 32-bit integers as 'I', 64-bit integers as 'J', single-precision floating-point numbers as 'F', double-precision floating-point numbers as 'D', and finally, 128-bit vectors as 'V'. The second problem is how to encode mutability. In SableWasm, we use capital letters for constant global variable types and lower letters for mutable ones. Finally, for function types, we follow a similar design as we used for global variables. SableWasm encodes a function type into a null-terminated string. Let's take `[i32, f32] -> [v128]` as an example. SableWasm encodes the type into '`IF:V`'. The colon acts as a separator between parameter types and result types. Note that '`:`' itself is also a valid SableWasm function signature string, and represents `[] -> []`, a void function with no arguments. Finally, metadata also encodes module names and entity names for import entities and names for export entities, which play a critical role later in the module initialization phase.

Function entity representation The WebAssembly specification classifies the functions into two groups, WebAssembly functions and host functions. WebAssembly functions are

[1] `https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html`

any functions defined within a WebAssembly module. On the other hand, host functions are directly provided by the host system, and from the WebAssembly module's perspective, the host functions are black boxes without any knowledge of their internals. Making things more complex, in MVP WebAssembly, there are no explicit requirements on how the WebAssembly functions should behave if they are invoked from other modules. Here we use a similar generalization like the one adopted by Javascript [2]. In short, in SableWasm, if a module exports a function, it exports the function in a closure that captures its enclosing instance. Suppose a second module invokes the exported closure as an import function. In this case, the function still only has access to its original module's entities and only communicates to the second module via return values. Hence, in SableWasm, we implement our function as a pair of pointers. The first one refers to its enclosing instance, and the second one relates to the generated function code. In this chapter's introduction, we mentioned that we pass the instance object as the first argument to the generated functions upon function calls. But, what should we give to the host function invocations? SableWasm defines that for all the host functions, the instance object pointer will always point to the caller's enclosing instance so that the host functions can access the internals of the caller's module.

Initialization protocol In the last part of the section, we will cover the initialization protocol we used in SableWasm. The initialization protocol consists of three basic steps: validation, instance preparation, and initialization. In the validation phase, we load the shared library with the operating system's help, such as `dlopen` on Linux, and check if it contains all the required symbols. Currently, a SableWasm shared library needs to export five symbols in total. Table 6.1 illustrates the symbols expected from the generated shared libraries. The instance initializer function takes a *prepared* instance object as the argument. The next step in SableWasm is to construct this *prepared* instance object. The idea of a *prepared* instance object is that we want to separate the memory allocation from the value

[2]WebAssembly Javascript Interface: `https://www.w3.org/TR/wasm-js-api-1/`

initialization. In SableWasm, the runtime system handles the memory allocation, while on the other hand, the initializer function takes care of the value initialization. In the second phase, the SableWasm runtime allocates all the entities and attaches them to the module instance. Note that SableWasm also resolves all the import names at this stage, and it will only proceed to the next step if all the expecting import entities are set. The import name binding utilizes the module names and entity names provided by the metadata. Finally, the last step is the initialization. SableWasm will invoke the initializer function supplied by the shared library. The initializer function takes care of all kinds of value initialization, such as setting values for global variables and copying data segments into linear memories. If the runtime system adequately prepares the instance context, the initializer function should never fail.

Symbol Name	**Description**
__sable_global_metadata	Metadata for global values
__sable_memory_metadata	Metadata for linear memories
__sable_table_metadata	Metadata for indirect tables
__sable_function_metadata	Metadata for functions
__sable_initialize	Instance initializer function

Table 6.1: SableWasm shared libraries exported symbols

6.2 WebAssembly Entities

In the previous section, we discuss the design of the SableWasm WebAssembly instance object. However, we treat all WebAssembly entities as opaque pointers without diving into the details during the last section. This section will cover the implementation of the WebAssembly entities along with the runtime library builtin functions in SableWasm. Before we start this section, we will first present the terms used throughout the later part of the thesis. In the rest of this chapter, we use `__sable_instance_t` to denote the type of the instance object. Similarly, we use a similar format when discussing WebAssembly linear memories, indirect tables, and global variables. For exam-

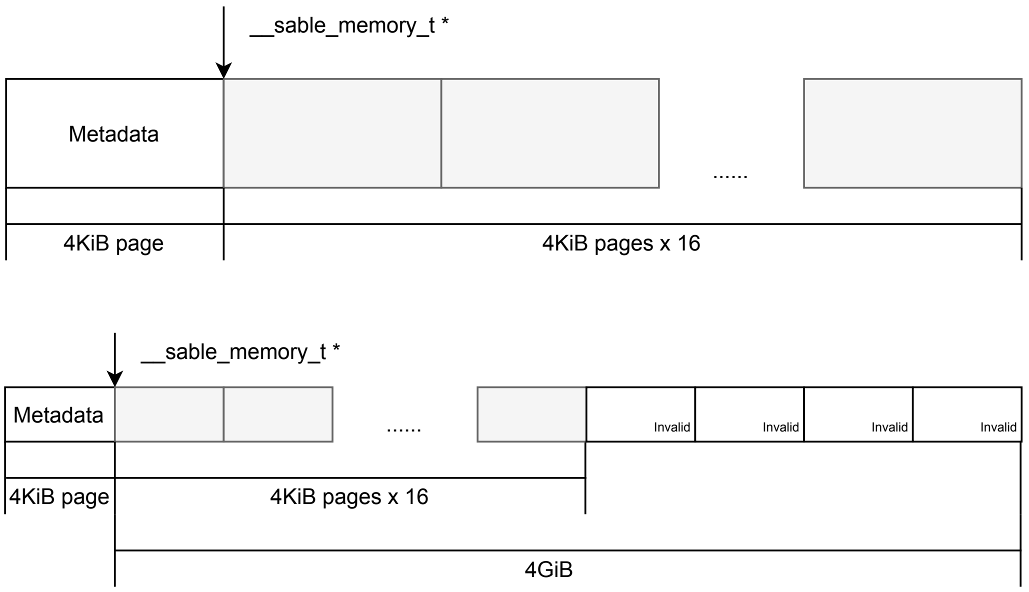


Figure 6.2: SableWasm WebAssembly linear memory

ple, `__sable_memory_t` is the type of WebAssembly linear memory in SableWasm. Finally, we use `__sable_function_t` refer to the function pointers that point to generated native functions.

Linear Memory SableWasm implements WebAssembly linear memories with mapped memory provided by the operating system. It also has a fallback implementation that uses standard `malloc` and `free` procedure from the C library for an operating system that does not support mapped memory. The fallback implementation is relatively trivial, and we will not discuss it in the thesis. Here, we will focus on the one that uses mapped memory. Figure 6.2 illustrates the strategies when mapping WebAssembly linear memory into native memory. On the top, we have a linear memory with a size of 1 in WebAssembly page size units, or 64KiB. In the figure, we assume the native machine has a page size of 4KiB, which is typical for most hardware architectures. Here's a quick recap on the requirements of WebAssembly linear memories. First, the program can efficiently random access any location within the linear memory. Second, at runtime, the module can query the size of the linear memory. Finally, the program can grow the linear memory if the runtime system allows it. SableWasm implements linear memories using a similar trick as the one used for 'malloc' functions in many C standard library implementations. From the generated shared libraries' perspective, the linear memory object points to the start of a continuous memory chunk. Hence, memory accesses are efficient and require only one layer of indirection. First, the generated function will fetch the linear memory base pointer from the instance object and calculate offsets accordingly. SableWasm also attaches an extra page that manages the metadata of the linear memory at the beginning. It contains all the records that the runtime system needs to work with the linear memory, such as the current size and the upper bound. Note that the size of the metadata is usually way smaller than the page size defined by the native machine. Still, SableWasm reserves a whole page for it, as we want our linear memory start address to be always page-aligned in the hope of better performance.

Runtime builtin functions	**Description**
__sable_memory_size	Query for the size of the linear memory
__sable_memory_guard	Perform boundary check on the linear memory
__sable_memory_grow	Attempt to increase the size of the linear memory

Table 6.2: SableWasm runtime builtin functions for linear memory

SableWasm implements additional functionalities through library functions. Table 6.2 illustrates all runtime library builtin functions provided by SableWasm. `__sable_memory_size` implements SableWasm's `MemorySize` instruction. It takes an argument of a linear memory instance and returns the size of it in WebAssembly page units. The second runtime builtin function, `__sable_memory_guard` corresponds to the `MemoryGuard` instructions in SableWasm. It takes a linear memory instance and the expected number of bytes ahead as arguments. Note that the function does not return any values, and this is intentional by design. SableWasm runtime library utilizes the C++ exception mechanism to report and handle errors. If the memory access is out-of-bound, the runtime system will throw an exception. We will come back to this later in this chapter when discussing the interaction between generated shared libraries and the host language. Finally, the last runtime builtin function, `__sable_memory_grow` implements the SableWasm's `MemoryGrow` instruction. The instruction follows its counterpart that appeared in the WebAssembly specification. It takes a linear memory instance and the number of pages to increase as arguments. If the operation is successful, the function will yield the new size of the linear memory; otherwise, it returns -1 instead. SableWasm grows the memory by remapping the memory with the help of the operating system. On Linux, this usually corresponds to a `mremap` operation.

In the above implementation, all linear memory bounds checks are explicit and program-directed, and thus they are relatively quite expensive. To further improve the performance, we use a similar technique to that used by many virtual machine implementations, which utilizes mapped memory access permission flags. Figure 6.2 illustrates this approach at the bottom. One may notice that MVP WebAssembly works with 32-bit addressing [3]. Hence, the maximum size of the linear memory is 4GiB. Thus, SableWasm reserves 4GiB of address when allocating a linear memory and marks all the pages beyond the current range as invalid pages. This operation is quite efficient as it only works with the memory address instead of allocating the memory. In this implementation, any out-of-bound access will result in a memory segmentation fault. Note that this strategy does yield better performance but results in a non-recoverable error. SableWasm provides both implementations, and one can select based on their needs. In the next chapter, when we compare SableWasm's performance against several other implementations, we always use the second strategy, as the recoverable code is not required.

Global Compared to the WebAssembly linear memory implementation, SableWasm's WebAssembly global variable implementation is relatively straightforward. In the current version of WebAssembly, global variables can only store primitive values. Therefore, SableWasm holds the WebAssembly global variable instance as a union construct of all possible value types, followed by its type's character encoding. Figure 6.3 illustrates

[3]This is subject to change in the future. WebAssembly 64-bit memory addressing: `https://github.com/WebAssembly/memory64`

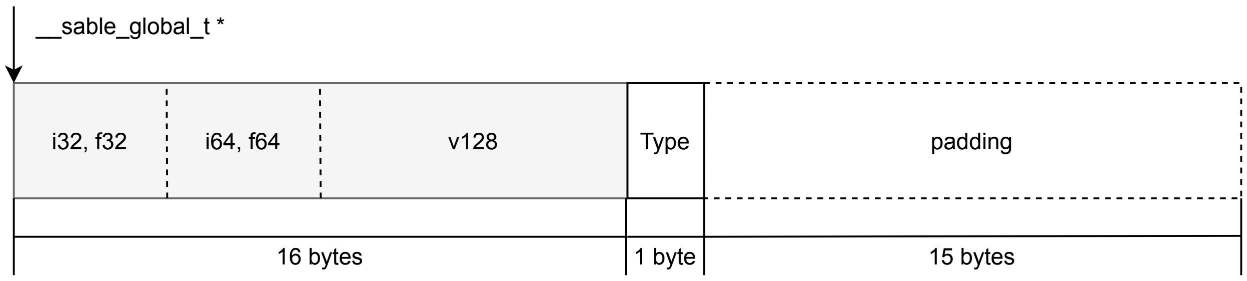


Figure 6.3: SableWasm WebAssembly linear global

SableWasm's implementation for WebAssembly global variable instances. From the generated shared libraries' perspective, the global variable access is equivalent to a simple load or store operation. Note that, in generated shared libraries, we never need to worry about the mutability of the global variables because the WebAssembly validation rules ensure that a valid module should never write to a constant global variable.

Indirect Table The last WebAssembly entity implemented in SableWasm is the indirect table, and it perhaps is the most complex one among all three of them. A quick reminder, in the instance layout section, we mentioned that we represent the function instance in SableWasm using function closures that capture its enclosing context. SableWasm implements the indirect table using a vector of function closures and their type signatures. The internals of the SableWasm indirect table is hidden from the generated shared libraries and only communicates to them via runtime buitlin functions. Table 6.3 illustrates all the runtime builtin functions provided by SableWasm for indirect tables.

Runtime builtin functions	**Description**
__sable_table_guard	Check if a given index is within the indirect table's range
__sable_table_check	Check if the entry has specific type
__sable_table_context	Fetch the context pointer of the entry
__sable_table_function	Fetch the function pointer of the entry
__sable_table_set	Write to indirect table

Table 6.3: SableWasm runtime builtin functions for indirect table

`__sable_table_guard` takes the indirect table instance and the index as arguments. It is quite similar to `MemoryGuard` instructions, except that it works with an indirect table. In addition, it also utilizes the same error handling strategy by throwing an exception in the case where the index is out-of-bounds. The next runtime builtin function, `__sable_table_check` implements the runtime type checking for indirect function calls. It takes a pointer to an indirect table instance, an index, and a function signature string as the parameter. We use the same strategy as we have seen in the instance layout section to encode the expecting type of the function. As in the current SableWasm, the type system is extremely trivial, there is no complex typing judgment involved, such as subtyping. Hence, the runtime type checking for indirect function calls is just a simple string comparison. In the case of type mismatch, the runtime type checking function also throws an exception. `__sable_table_context` and `__sable_table_function` are the getter functions provided by SableWasm. Both of them take an indirect table instance and an index as the argument. These two functions assume the access is within range, and the indirect table entry has the expected function type. We will come back to this later in the chapter when we discuss the patterns used when lowering SableWasm MIR into LLVM intermediate representations. As their names suggest, the first function returns the pointer to the context instance object, and the second function returns the function code address. Finally, the last runtime builtin function for indirect tables is `__sable_table_set`. Although in MVP WebAssembly, indirect tables are immutable, the program cannot alter them after they initialized [4], the module initialization function still needs the setter function to setup WebAssembly element segments. The setter function takes an indirect table instance, an index, a function code address, and its null-terminated type signature string as the argument. Similar to the getter functions, the setter function assumes the index is always within range.

The SableWasm runtime library still provides another runtime builtin function that does not fit into the categories above. SableWasm MIR defines an `Unreachable` instruc-

[4]This is subject to change in the future. WebAssembly reference types: `https://github.com/WebAssembly/reference-types`

tion, which should never reached by any control flow, and if so, it will signal a runtime panic. In many other languages, `Unreachable` maps to a hardware trap instruction, such as `ud2` instruction on x86 architecture. However, this behaviour is not acceptable in SableWasm. `ud2` generates a non-recoverable hardware invalid instruction exception, which will eventually lead to the entire system core dump; on the other hand, SableWasm expects exceptions thrown from generated shared libraries and should handle them accordingly. Hence, the SableWasm runtime library provides the `__sable_unreachable` function for the SableWasm MIR `Unreachable` instruction. We will come back to this with more details in the following section when discussing the code generation strategy used when lowering SableWasm MIR into LLVM intermediate representation.

6.3 Code Generation

This section describes the code generation strategy used in the SableWasm LLVM backend. For most of the instructions, especially for SableWasm MIR numeric operations, the translation rules are simple mappings between SableWasm MIR instructions and their LLVM counterparts. In this section, we will skip the discussion over these trivial mapping. Instead, one can consult the SableWasm source code for more details. The rest of the section will focus on several key aspects: local variable implementation, linear memory manipulation, indirect function calls, and SIMD instruction operations. One problem that arises when lowering SableWasm MIR into LLVM intermediate representation is how to pick the instruction translation order. Any instruction in SableWasm MIR can refer to values either generated by a previous instruction in the same basic block or instructions within a dominating block, implying that when lowering SableWasm MIR, we need to perform a pre-order tree traversal over the dominator tree. However, ϕ nodes may exist merging candidate values from prior dominating blocks or due to subsequent backward branching. Hence, the translation visitor may not have translated the candidate value before ϕ nodes. SableWasm backend takes a two-phase translation to address this problem.

In the first pass, the backend will translate all the instructions and collect the resulting values into a map, and in the second pass, the backend will come back to the ϕ nodes and fix up the candidate values accordingly.

Function declaration and local variables

```
function %foo: [i32] -> [f32] {
  {(arg) %local0: i32, %local1: f64}
  ......
}
⟹
define private float @foo(%__sable_instance_t* %0, i32 %1) {
entry:
  %2 = alloca i32, align 4
  store i32 %1, i32* %2, align 4
  %3 = alloca double, align 8
  store double 0.000000e+00, double* %3, align 8
  ......
}
```

```
{%local: i32}
%t0 = local.get %local ⟹ %t0 = load i32, i32* %local, align 4
local.set %local %t0 ⟹ store i32 %t0, i32* %local, align 4
```

We will first start by examining the translation pattern for lowering SableWasm MIR functions into LLVM functions and their local variables. The example above presents a simple function named `foo`, which takes a single 32-bit integer as the argument and returns a single-precision floating-pointer number. `foo` has two local variables. The parameter implicitly introduces the first one, `local0`, and the function explicitly defines the second one, `local1`. At runtime, `local0` will hold the value of the parameter upon entry, and `local1` will initialize to zero. Compared to the SableWasm MIR function definition, the one in LLVM intermediate representation (IR) has three major differences. First, the LLVM function definition has the extra instance object pointer in the arguments, in the example above, `%0`. We covered this briefly in the instance layout section. In short, for all the functions, the SableWasm backend code generator will implicitly add the instance object pointer as the first argument. The second difference is in the entry block. SableWasm MIR, similar to WebAssembly, views the local variables as opaque memory slots.

However, LLVM IR requires users to manually allocate them in stack memory space via the `alloca` instruction. The `alloca` instruction reserves enough memory on the stack based on the given type and returns a pointer. In example above, `%2` and `%3` are two reserved local variable memory region that correspond to `local0` and `local1` accordingly. The last difference is that SableWasm IR defines implicit initialization for all local variables; on the other hand, LLVM `alloca` instruction leaves the reserved memory with uninitialized values. Hence, to faithfully implement WebAssembly and SableWasm MIR specification, we generate `store` instructions to set the initial values for each local variables. As for `LocalGet` and `LocalSet` instructions, the translation patterns are quite straightforward. The SableWasm backend code generator maps `LocalGet` instructions to `load` instructions and `LocalSet` instructions to `store` instructions as demonstrated in the example above.

Linear memory operation

```
Fetching linear memory:
%t0     = getelementptr
            inbounds %__sable_instance_t, %__sable_instance_t* %0,
            i32 0, i32 4
%memory = load %__sable_memory_t*, %__sable_memory_t** %t0, align 8

%t0 = memory.size %mem ⟹
%t0 = call i32 @__sable_memory_size(%__sable_memory_t* %mem)
%t0 = memory.grow %mem %delta ⟹
%t0 = call i32 @__sable_memory_grow(%__sable_memory_t* %mem, i32 %delta)
memory.guard %mem %offset ⟹
call void @__sable_memory_guard(%__sable_memory_t* %mem, i32 %offset)
```

In section 6.1 and 6.2, we presented how the instance object manages the linear memory instance and several runtime functions that implement additional functionalities. The SableWasm backend code generator takes advantage of the design by mapping SableWasm linear memory manipulation instructions into builtin function invocations. The example above demonstrates the mapping for `MemorySize`, `MemoryGrow` and `MemoryGuard` instructions. All these instructions map to `call` instructions to their corresponding builtin functions with appropriate arguments. Note that all builtin functions require passing the linear memory pointer as an argument. Currently, the WebAssembly module can have at most one linear memory. Due to the validation rules, such linear memory must present within the module if linear memory manipulation instructions appear in the program. Further, as we store linear memory instance pointers before any other entities, one can show that the linear pointer must be the 5th pointer in the instance object. Hence, the SableWasm backend code generator fetches the linear memory instance pointer using a pair of a `getelementptr` instruction and a `load` instruction. The `getelementptr` instruction LLVM calculate addresses for entries in a aggregation. The above example calculates addresses base on the type `__sable_instance_t` which is generated according to declared entities at compile time.

Linear memory load and store

```
Load a 32-bit integer:
%result = load.32 i32 %mem %addr ⟹
  %t0     = ptrtoint %__sable_memory_t* %memory to i64
  %t1     = zext i32 %offset to i64
  %t2     = add nuw i64 %t0, %t1
  %addr   = inttoptr i64 %t2 to i32*
  %result = load i32, i32* %addr, align 1
Partial load a 32-bit integer:
%result = load.16 i32 %mem %addr ⟹
  ......
  %t0     = load i16, i16* %addr, align 1
  %result = zext i16 %t0 to i32
Store a 32-bit integer:
store.32 %mem %addr %val ⟹
  ......
  store i32 %val, i32* %addr, align 1
Partial store a 32-bit integer:
store.16 %mem %addr %val ⟹
  ......
  %t0    = trunc i32 %val to i16
  store i16 %t0, i16* %addr, align 1
```

SableWasm MIR classifies load and store instructions into two groups, partial and complete. A quick reminder, WebAssembly associates load and store operations with sign extension mode, while in SableWasm, we define load instruction to perform zero extension, and store instructions always apply bit truncation. The first example above presents a complete load operation for a 32-bit integer. The translation pattern is relatively straightforward. Note that the linear memory instance pointer points to the first byte within the linear memory. Hence, the SableWasm backend code generator will first calculate the native write address by summing up offset and base pointer and map the `Load` instruction to `load` in LLVM. The LLVM memory operation, such as `load` and `store` has a complementary attribute, `align`. In the background section, we introduced the attributes in LLVM. In short, the `align` attribute marks an alignment requirement for memory access operations. As WebAssembly linear memory is comparable to a byte array, in which read-write can occur at any point, we can only conservatively set the alignment to one in order to limit the LLVM backend instruction selector from generating instructions with alignment assumptions. This, in theory, leads to less efficient code. However, later in the evaluation section, we determine this is not a bottleneck of the entire implementation. In the future, one can further improve the performance of SableWasm by designing analyses that infer lower bounds for alignment. The second example above demonstrates the translation pattern for partial load operation. Compared to the complete load instruction, the translation pattern for partial load instruction has an additional zero-extending operation, `zext` at the bottom, to implement the SableWasm MIR partial load semantics. On the other hand, the translation pattern for both complete and partial `store` instructions are very similar to `load` instructions. The most notable difference is the `trunc` instruction in partial `store`'s translation pattern which performs bit truncation on the operand.

Indirect function call

```
call.indirect %table %index %expect_ty ⟹
  call void @__sable_table_guard(%__sable_table_t* %table, i32 %index)
  call void @__sable_table_check(
    %__sable_table_t* %table, i32 %index, i8* %expect_ty)
  %t0 = call %__sable_instance_t* @__sable_table_context(
    %__sable_table_t* %table, i32 %index)
  %t1 = call %__sable_function_t* @__sable_table_function(
    %__sable_table_t* %table, i32 %index)
  %t2 = icmp eq %__sable_instance_t* %t0, null
  %t3 = select i1 %t2, %__sable_instance_t* %0, %__sable_instance_t* %t0
  %t4 = bitcast %__sable_function_t* %276 to ......
  %t5 = call ...... %t4(%__sable_instance_t* %t3, ......)
```

The SableWasm backend code generator implements indirect function calls via a series of builtin function invocations. We have already presented the builtin function in section 6.2; hence, we will not show them in detail in this paragraph. The first step for calling an indirect function is to check if the index is within range by calling the `__sable_table_guard` builtin function. If the index is within range, we then compare the expected function type with the actual indirect function type with `__sable_table_check`. Note that this builtin function also checks if the entry is a null function. If so, it will report an exception. The SableWasm backend code generator uses a similar technique to encode the expected function type into a null-terminated string, as we have seen in section 6.1. After we make sure the indirect function is valid, we can now fetch the context pointer and function address pointer by using two getter functions, `__sable_table_context` and `__sable_table_function`. Before we invoke the function, we need to check if it is a host function. A quick reminder, SableWasm will set context pointers for all host functions as null pointers, and when invoking a host function, we need to pass the current instance object pointer as the context pointer. The SableWasm code generator chooses the correct context pointer by using a pair of `icmp` and `select` instruction. After selecting the correct context pointer, the indirect function is straightforward by casting the function code address into the function pointer and invoking it appropriately. One may notice that the indirect function call in SableWasm is costly and involves multiple function calls. WebAssembly specification does not impose requirements on indirect function call efficiency, and later in our benchmark, we determine that indirect function calls are not a performance bottleneck. Hence, the SableWasm code generator focus on extensibility rather than performance.

SIMD operation The last translation pattern we will cover in the section is the SIMD operations. For most of the SIMD operations, the SableWasm backend code generator maps to their LLVM counterparts. However, one challenge arises when translating SableWasm

MIR into LLVM intermediate representation around the type system. In section 5.2.3, we presented the type system for SableWasm MIR. A quick reminder, the SableWasm MIR follows WebAssembly's design by erasing the shape information from the vector values, depending on instructions to interpret them correctly. However, LLVM intermediate representation does require shape information for vectors. Hence, when lowering SableWasm MIR into the LLVM intermediate representation, the SableWasm backend code generator needs to insert cast instructions when required. For most of the numerical instructions, this is pretty trivial. The backend code generator will first infer an LLVM vector type based on the SableWasm instruction shape information. For example, `v128.add i16x4` implies that the operand must have type `<4 x i16>` in LLVM. In the case where the shape type is unsuitable, the SableWasm backend code generator will insert a bit cast, `bitcast to`. The bit cast operation is always valid as, in the current version of SableWasm MIR, we only work with 128-bit vectors. However, there are still several corner cases in this strategy. What type should we assign to ϕ nodes when merging vectors from multiple control-flow? Also, what type should we assign for load instruction when shape information is still not yet available? The SableWasm backend code generator takes advantage of the fact that integer types in LLVM can be arbitrarily long, and more specifically, 128-bit integer, `i128`, is a valid type in LLVM. The SableWasm backend code generator will always use `i128` as a default type in these corner cases. For example, for load instruction for SableWasm vectors, the code generator will emit a `load` instruction with `i128` type, and later when any instruction takes the value as the operand, it will setup the bit cast instruction accordingly.

6.4 Interface with C/C++

The last section of this chapter will cover the interface between the generated shared library and the host languages. Currently, SableWasm only has a binder library for C/C++. However, the principle is relatively straightforward, and one can add implementations of the binder function for any other language. In the rest of the section, we will focus our discussion on the callee wrapper, WASI function implementations and error handling strategies.

Callee wrapper Section 6.1 mentioned that SableWasm stores a function instance as a pair of context pointer and function address pointer. Additionally, SableWasm also encodes the function types as null-terminated strings. However, all this information is only available to the host program at runtime. C/C++ is a statically typed language; hence, we can only specify type contracts on the exported functions at compile-time and verify the contracts at runtime. Traditionally, one can use a type erased pointer, a `void` pointer, to store the function address and reinterpret it to the actual concrete type. SableWasm presents a helper class that provides type-safe access to the exported functions: `WebAssemblyCallee`. `WebAssemblyCallee` takes advantage of the template metaprogramming system in C++ and generates a null-terminated encoding of an expected type at compile-time. At runtime, the wrapper class will check the type signature string against the actual type string before forwarding the function call. If the type signature string does not match, the system will signal an exception.

WASI interface implementation WebAssembly System Interface (WASI) extends WebAssembly by providing syscalls that interact with the host environment. This extension is non-invasive, and all the syscalls are in the form of imported functions, mainly host functions. Hence, SableWasm implements the WASI extension using host library functions only. At the shared library initialization phase, the loader will set up WASI host functions based on the import descriptor. Currently, SableWasm only implements the minimal WASI interface functions necessary in order to run benchmarks, such as standard I/O and timing. However, the framework is easy to extend, and all the WASI function implementations are under the namespace `runtime::wasi`. Therefore, we will skip them in detail in the thesis; one can consult the source code for implementation details of WASI interface functions. One of the project's future work is to continuously work on the

```
using namespace runtime;

int run(std::filesystem::path const &Path) {
  WebAssemblyInstanceBuilder InstanceBuilder(Path);

#define WASI_IMPORT(name, func)                                   \
  InstanceBuilder.tryImport("wasi_snapshot_preview1", name, func)
  WASI_IMPORT("proc_exit"     , wasi::proc_exit     );
  WASI_IMPORT("clock_time_get", wasi::clock_time_get);
  WASI_IMPORT("args_sizes_get", wasi::args_sizes_get);
  ......
#undef WASI_IMPORT

  WebAssemblyInstance Instance = InstanceBuilder.Build();
  WebAssemblyCallee FnStart = Instance->getFunction("_start");
  try {
    FnStart.invoke<void>(); // _start : [] -> []
  } catch (wasi::exceptions::WASIExit const &Exception) {
    return Exception.getExitCode();
  }
  return 0;
}
```

Figure 6.4: Simple C++ SableWasm loader function

WASI system interface and add more features to SableWasm, such as capability-based file system and networking.

Error handling strategies The last topic we will address in the section is error handling. SableWasm builds its error handling strategy based on the C++ exception mechanism. Comparing to other exception handling strategies, this brings us two significant benefits. First, when generating LLVM intermediate representation for shared libraries, we can avoid boilerplate code that propagates exceptions. Additionally, on most modern system ABIs that supports zero-cost exception handling, this gives SableWasm a performance advantage. On the other hand, this leaves us room for further improvement for pending WebAssembly extensions, such as the WebAssembly exception handling extension [5]. The WebAssembly exception handling extension generalizes the WebAssembly specification by adding `try catch` construct to the syntax, which directly corresponds to the C++ exception handling mechanism.

[5]WebAssembly exception handling: `https://github.com/WebAssembly/exception-handling`

In this section, we discussed the interaction between C/C++ and the SableWasm system. We will conclude the chapter with a concrete loader function example. Figure 6.4 demonstrates a simple loader function for generated SableWasm shared libraries. In the example above, we assume the WebAssembly module is a WASI compatible module, and hence, exports a function named `_start` as the entry function with type `[] -> []`.

Chapter 7

Evaluation

In the previous chapters, we presented the design of the SableWasm compiler and runtime. This chapter will focus on the performance evaluation in terms of the execution speed of the generated shared libraries. Here, we focus on three research problems. First, how does SableWasm perform compared to other WebAssembly runtime environment implementations? Second, does the optimization over the input WebAssembly module affect the overall performance? Finally, how much does the WebAssembly SIMD extension improve comparing to optimized scalar counterparts? We will first present the setup for experiments used when investigating three questions, and later, the experimental results for each one of them.

7.1 Experiment Setup

This section presents the setup for the experiments in the remaining part of the chapter. We conduct the benchmarks on the same server for all experiments. The experiments were performed on a six-core Intel Core processor at a 3.7 GHz standard clock frequency and with an L3 cache of 12 MiB. Additionally, the server runs Ubuntu 18.04 with Linux kernel version 4.15.0 and 32GiB of memory. When measuring the performance, we execute each benchmark ten times in succession to minimize the measurement error as some

Benchmark Name	Description
2mm	2 matrix multiplication (D = A.B; E = C.D)
3mm	3 matrix multiplication (E = A.B; F = C.D; G = E.F)
adi	alternating direction implicit solver
atax	matrix transpose followed by vector multiplication
bicg	BiCG sub kernel of BiCGStab linear solver
cholesky	Cholesky decomposition
correlation	correlation computation
covariance	covariance computation
doitgen	multiresolution analysis kernel (MADNESS)
durbin	Toeplitz system solver
dynprog	dynamic programming (2D)
fdtd-2d	2D finite different time domain kernel
fdtd-apml	FDTD using anisotropic perfectly matched layer
gauss-filter	gaussian filter
gemm	matrix-multiply (C = alpha.A.B + beta.C)
gemver	vector multiplication and matrix addition
gesummv	scalar, vector and matrix multiplication
gramschmidt	Gram-Schmidt decomposition
jacobi-1D	1D Jacobi stencil computation
jacobi-2D	2D Jacobi stencil computation
lu	LU decomposition
ludcmp	LU decomposition (different implementation)
mvt	matrix vector product and transpose
reg-detect	2D image processing
seidel	2D Seidel stencil computation
symm	symmetric matrix multiplication
syr2k	symmetric rank-2k operations
syrk	symmetric rank-k operations
trisolv	triangular solver
trmm	triangular matrix multiplication

Table 7.1: the Polyhedral benchmark suite (Polybench)

of the benchmarks take less than a second to complete. Finally, the final benchmark result is the average among ten runs except the highest and the lowest. For the benchmark subject, we choose three different benchmark suits, the Polyhedral benchmark suite (Polybench), the Ostrich benchmark suite (Ostrich), and the NAS parallel benchmarks (NPB).

Benchmark Name	Description
back-prop	backward propagation in a layered neural network
bfs	breadth-first search in a randomly generated graph
crc	CRC error-detecting algorithm
fft	fast Fourier transform
hmm	forward-backward algorithm over a hidden Markov model
lavamd	3D space particle simulation
lud	LU decomposition
nqueens	N-queen problem solver
nw (needle)	find optimal alignment of two protein sequences
page-rank	page-rank algorithm to measure the importance of a web site
spmv	sparse matrix multiplication with a vector
srad	diffusion method for ultrasonic and radar imaging

Table 7.2: the Ostrich benchmark suite (Ostrich)

Polybench The Polyhedral benchmark suite (Polybench) [36] contains a group of small math kernel functions as shown in table 7.1. The description table is adjusted from official Polybench documentation [1]. In the WebAssembly announcement paper [8], the community also chose Polybench as the evaluation subject. However, one problem is that the Polybench is in C. Therefore, the researchers cross-compiled the benchmark using a modified Clang compiler with an LLVM WebAssembly backend. However, there is no standardized system interface, such as WASI, proposed by the community when publishing the paper. Hence, the experiment is measured with an external clock, and all features that require system interaction are disabled. On the other hand, when evaluating SableWasm, we use a WASI-enabled [2] Clang compiler to cross-compile the WebAssembly modules into WebAssembly modules. Each benchmark reports its execution time by issuing syscalls to the runtime environment, which in theory, should yield more accurate results, especially for a just-in-time (JIT) runtime environment.

Ostrich The second benchmark suite we used was the Ostrich benchmark suite [9], illustrated in table 7.2. Comparing to the Polybench, Ostrich focuses on larger scientific

[1]Polybench: `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`
[2]WASI SDK: `https://github.com/WebAssembly/wasi-sdk`

Benchmark Name	**Description**
IS	integer sort (bucket sort)
EP	Marsaglia polar method for generating random numbers
CG	estimate the smallest eigenvalue of a SPD matrix
MG	multi-grid on a sequence of meshes
FT	fast Fourier transform
BT	block tri-diagonal solver
SP	scalar penta-diagonal solver
LU	lower-upper solver

Table 7.3: the NAS parallel benchmark suite (NPB)

problems instead of computation kernels. The Ostrich benchmark suite supports multiple programming languages, such as Javascript and C. Here, we prepare the WebAssembly module similar to the Polybench benchmark suite with a WASI-enabled Clang compiler. However, unlike the Polybench benchmark suite, which does not require any modification on the source code, we need to tweak the Ostrich benchmark code due to the limitations of WebAssembly specification. This includes hard-coding the command-line arguments and replacing throwing an exception with calling the `exit` function.

NPB The last benchmark suite we selected for evaluating SableWasm is the NAS parallel benchmark suite [33], shown in the table 7.3. We choose this benchmark because of its parallel nature, as the third research question focuses on the SIMD instruction operations. However, the original NPB benchmark suite is in Fortran, and, at the time of thesis writing, there is no cross-compiler from Fortran to WebAssembly. Hence, we choose an OpenMP variant instead [3]. Although the currently WASI-enabled Clang does not support OpenMP, we can still cross-compile into WebAssembly, as OpenMP code trivially reduces to C.

This section presents the benchmark environment and test cases for the experiments later in the chapter. One may notice some duplication among three benchmark suites, such as the upper-lower matrix decomposition (LU, ludcmp) and fast Fourier transform

[3]NPB OpenMP C: `https://github.com/benchmark-subsetting/NPB3.0-omp-C`

(FT, fft). However, we will still treat them as different individual test cases for all of them, as they come with various implementations and may lead to performance differences. Another problem that arises when preparing WebAssembly modules for benchmark suits is that some of the generated modules from the WASI-enabled Clang compiler have unexpected behaviour. In NPB, although the WASI-enabled Clang compile can successfully translate all test cases for all eight benchmark cases, there are two among eight test cases that have different behaviour compared to their native counterparts. For example, the WebAssembly module for the IS benchmark case has a memory access out-of-bounds error for native and optimized translation. Also, the module for EP failed when compiled with the optimization flag enabled in the WASI-enabled Clang compiler. We suspect that some unknown bugs in the toolchain may exist as it is still under active development. Another possible cause for the problem is that the OpenMP implementation may contain non-standard operations that result in undefined behaviour. We also test the generated modules against several other WebAssembly runtime environments, and the result is consistent. The last problem we encountered during benchmarking is around WebAssembly SIMD operation extensions. As the extension is still under standardization, most runtime environments only support a subset of all instructions. Hence, when comparing SIMD operations, some of the benchmark results are infeasible. However, we still manage to collect SIMD operation performance data for most of the benchmark cases.

7.2 RQ1: How does SableWasm perform compare to others?

This section will compare SableWasm performance against several other WebAssembly runtime environments, specifically Wasmtime and Wasmer. We will benchmark three implementations over naive (`-O0`), optimized (`-O3`), and SIMD-enabled optimized (`-O3 -msimd128`) WebAssembly modules compiled from the source. One can consider Wasmtime [4] as the 'reference' implementation of WebAssembly out of the browser and it is

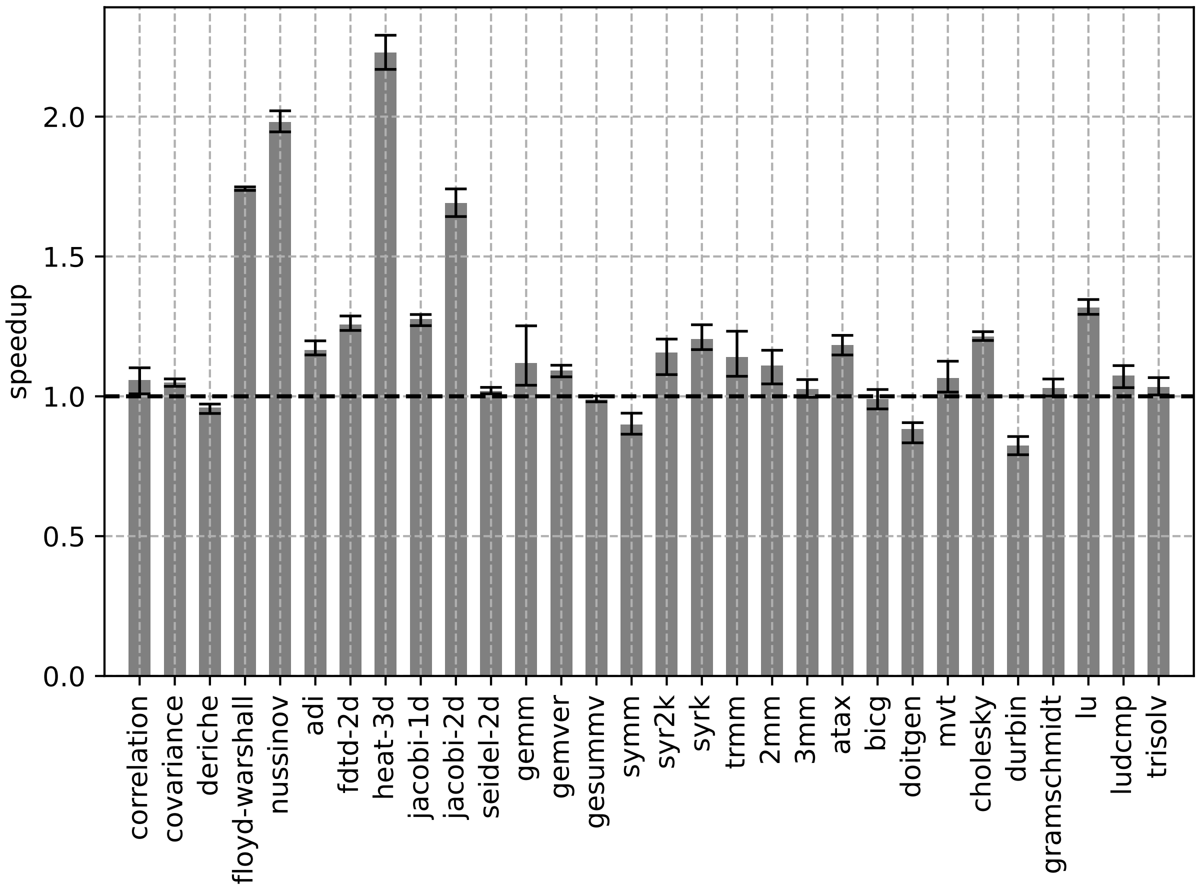
[4]Wasmtime: `https://github.com/bytecodealliance/wasmtime`

maintained by the WebAssembly community group. The system is built upon the custom compile framework, Cranelift [5]. Currently, both Cranelift and Wasmtime are still under active development and subject to changes in the future. Here, in this project, we anchor our Wasmtime at version 0.26.0. Wasmer [6] is another community approach for running WebAssembly sandboxed applications outside of the browser. It comes with a package manager, WAPM [7], that distributes applications in WebAssembly binary format. Wasmer supports three compiler backends, LLVM, Cranelift, and a single-pass code generator for fast compilation. In this chapter, we will focus on the LLVM and Cranelift variants of Wasmer. Similar to Wasmtime, Wasmer is also under active development at the time of thesis writing, and we fix the version of Wasmer at 1.0.2. Unlike SableWasm, an ahead-of-time (AOT) compiler for WebAssembly modules, Wasmtime and Wasmer are both just-in-time (JIT). Thus, when measuring the benchmark's performance, we need to isolate the error induced by the compiler, such as compilation-overhead and warm-up time. To eliminate the compilation-overhead, we measure the execution time with the internal timing code by issuing syscalls to the WASI layer. Further, we adjust the benchmark size for Ostrich and NPB so that each benchmark case takes more than 10 seconds to compute to reduce the error introduced by the JIT warm-up process.

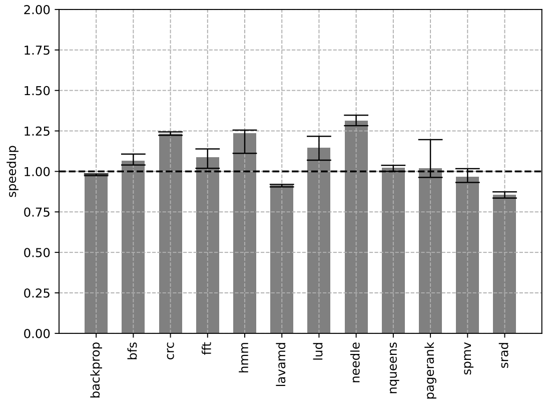
Figures 7.1 (page 101) to 7.9 (page 110) present the benchmark results. We normalize the data with respect to the SableWasm's execution time and present them as speedups. A number higher than one means that the SableWasm's performance is better than the candidate, and on the other hand, a less than one speed-up refers to slow-down. The error bar is calculated based on the 10th percentile and 90th percentile accordingly.

For naive translated WebAssembly modules, shown in figures 7.1 to 7.3, SableWasm performs better than Wasmtime in most benchmark cases except seven of them. We suspect that the slow-down comes from the excessive linear memory access. In the current version of the WASI-enabled Clang compiler, a naive translated module will use linear

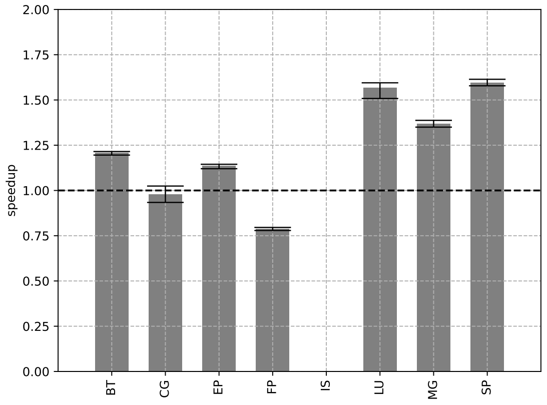
[5]Cranelift: `https://github.com/bytecodealliance/wasmtime/tree/main/cranelift`
[6]Wasmer: `https://wasmer.io/`
[7]WAPM: `https://wapm.io/`



(a) Polybench

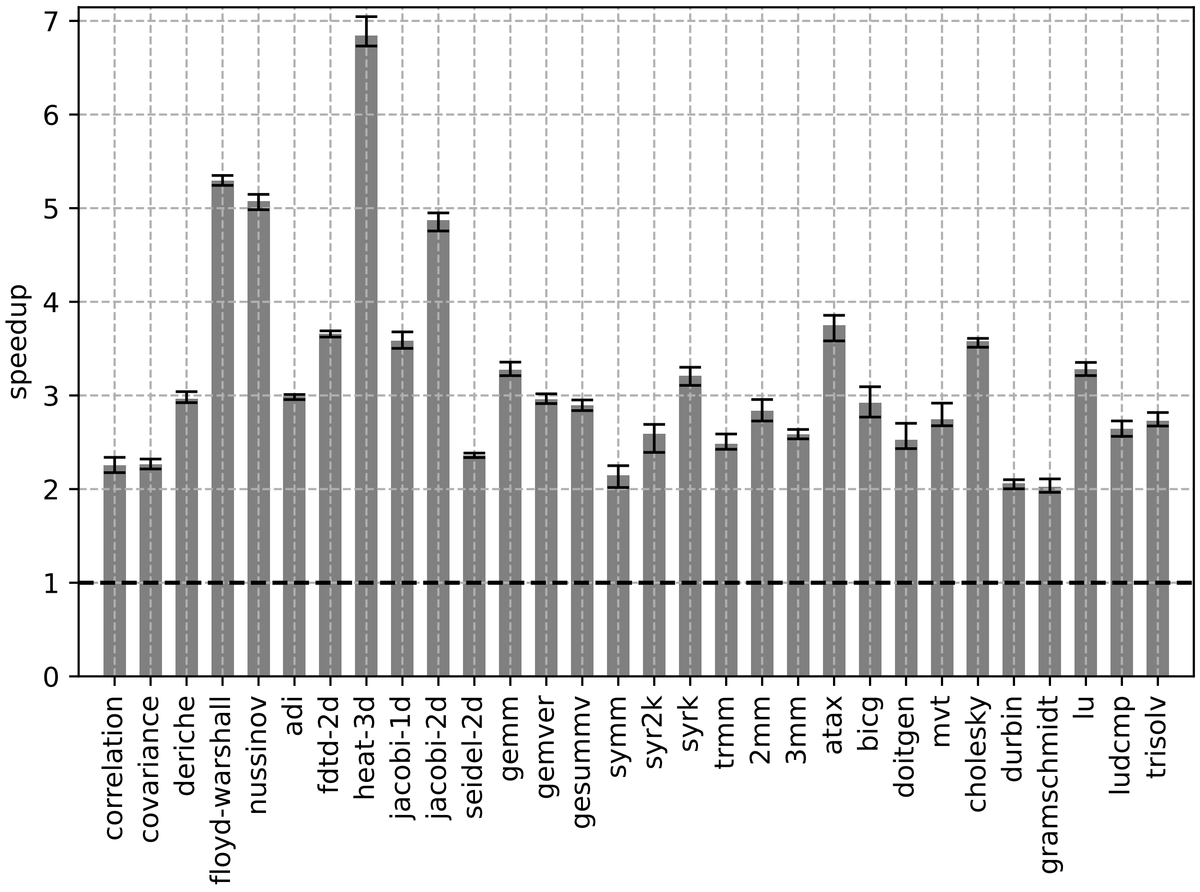


(b) Ostrich

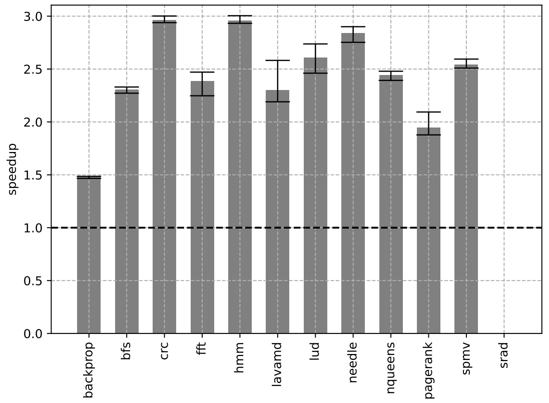


(c) NPB

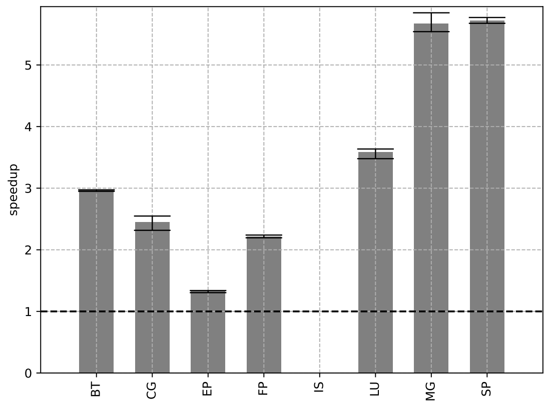
Figure 7.1: Benchmarks with naive (`-O0`) on Wasmtime



(a) Polybench

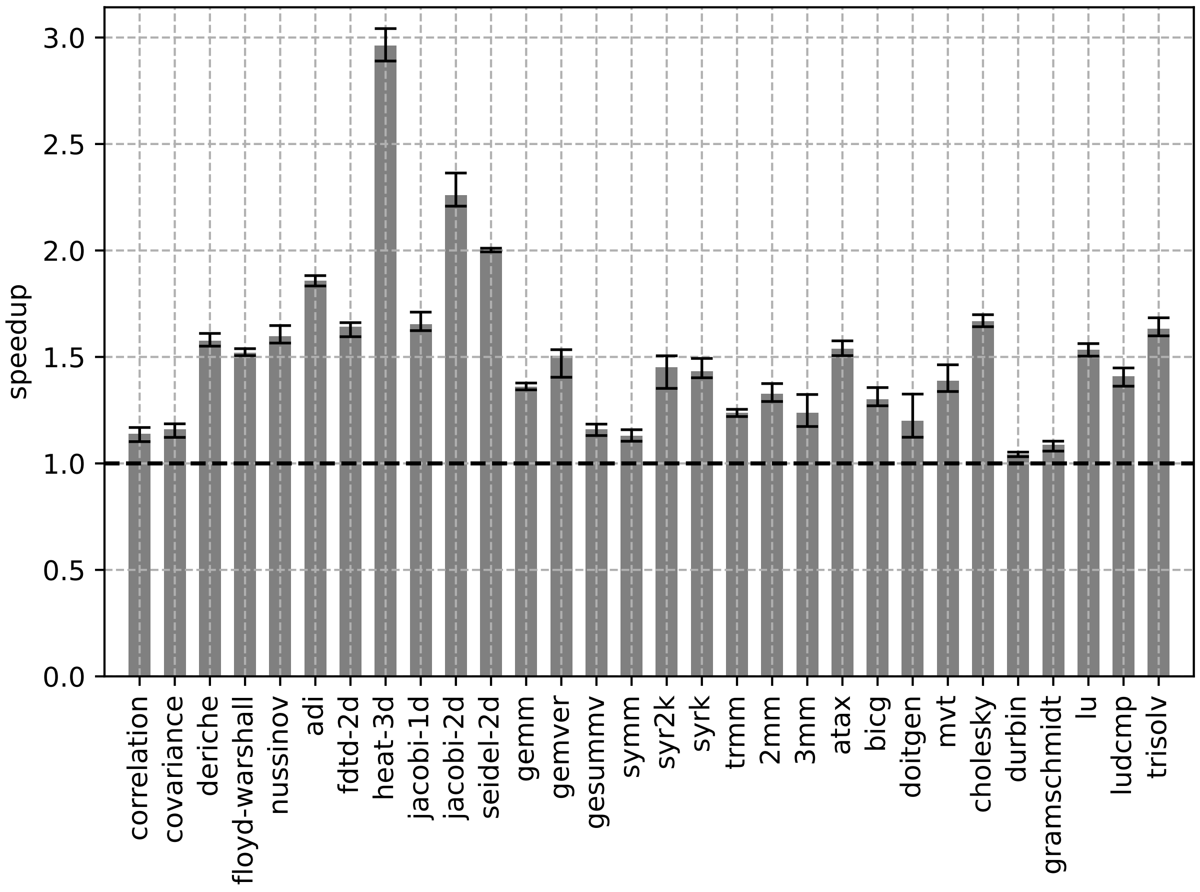


(b) Ostrich

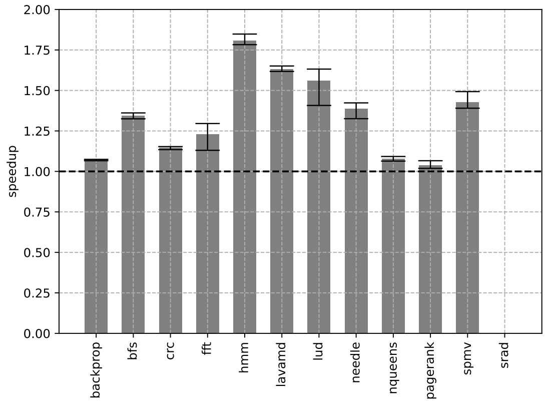


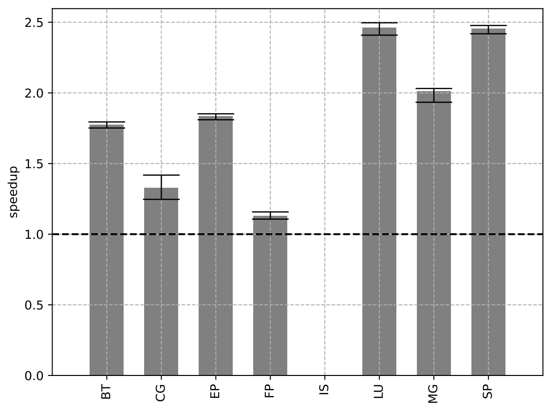
(c) NPB

Figure 7.2: Benchmarks with naive (`-O0`) on Wasmer (Cranelift)



(a) Polybench





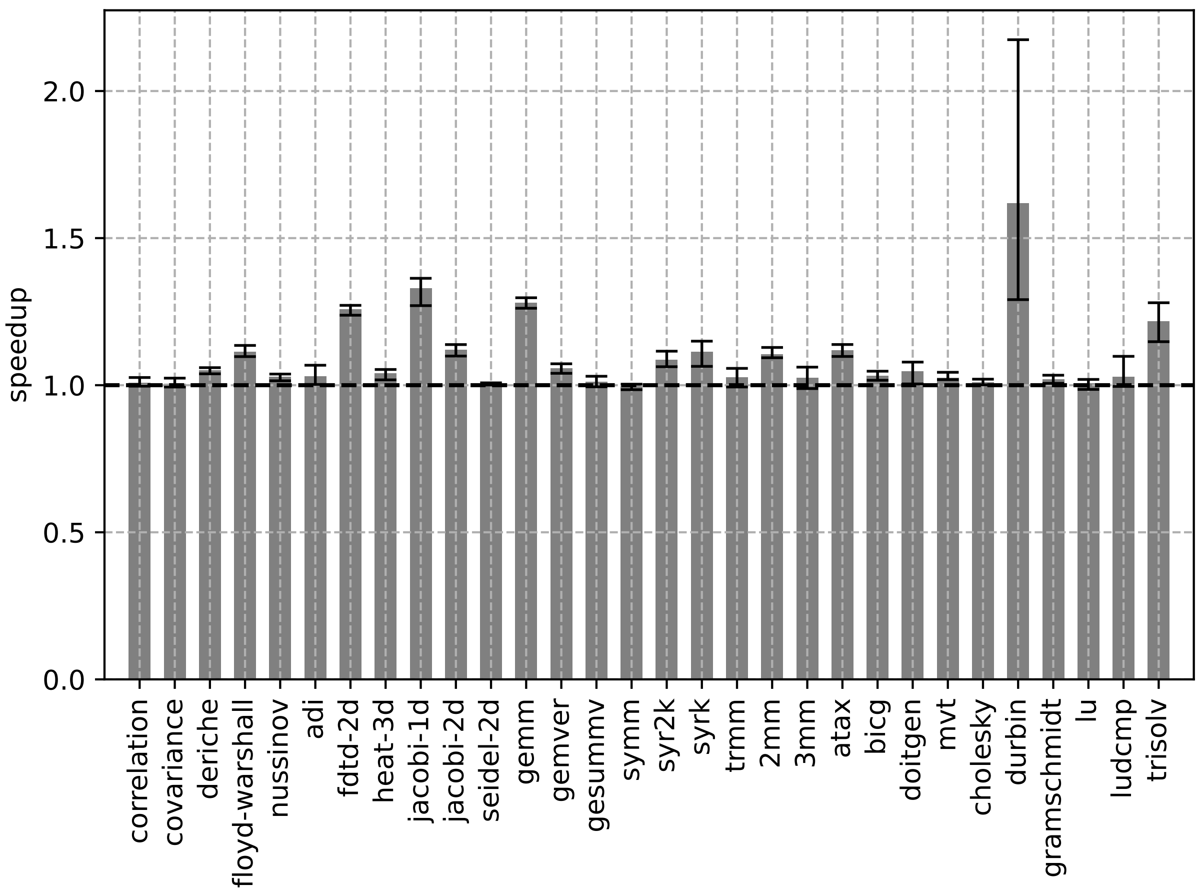
(b) Ostrich

(c) NPB

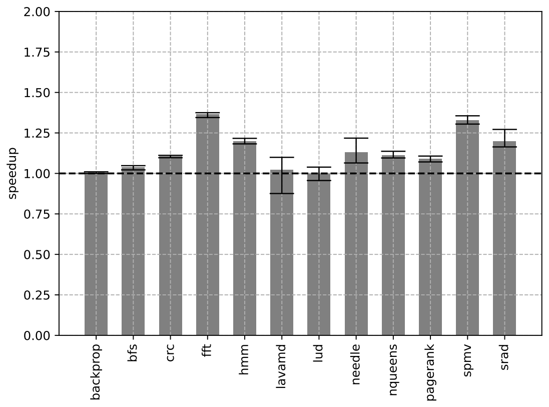
Figure 7.3: Benchmarks with naive (`-O0`) on Wasmer (LLVM)

memory to simulate stack frame functions instead of using local variables. This means that when writing to a function's local variable, SableWasm needs to first load the linear memory base pointer from the instance object, calculate the address, and then perform the memory access. Making the case worse, the current SableWasm will always load the base memory pointer even if a local variable already holds the base pointer. LLVM cannot effectively eliminate these load instructions, as the linear memory base pointer in the instance object is volatile. One possible solution to ease the problem is to carefully annotate the instance object pointer so that the alias analysis in LLVM can correctly identify these redundant load instructions. On the other hand, SableWasm performs better than Wasmer with both Cranelift or LLVM backend. This is quite interesting as the current SableWasm is also built upon LLVM. We suspect that two factors are contributing to the speedup. First, SableWasm employs several optimization techniques to improve the quality of the generated LLVM intermediate representation. When designing the translation patterns for lowering SableWasm MIR into LLVM IR, we notice that the quality of LLVM IR has a significant impact on the result performance, especially for auto-vectorization. Second, Wasmer supports many other safety features that are not specified in the WebAssembly specification, such as stack probing. These safety features impose performance drawbacks which may also contribute to the performance difference.

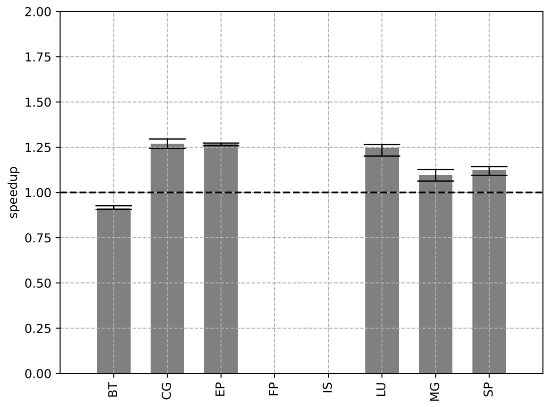
For optimized and SIMD-enabled input WebAssembly modules, show in figures 7.4 (page 105) to 7.9 (page 110) SableWasm performs on par with Wasmtime, except on benchmark case `durbin`. One may also notice that the error for `durbin` in figure 7.4a is more significant compared to others. This is due to the nature of the `durbin` benchmark case. The `durbin` benchmark contains a tiny computation kernel function and only takes a few milliseconds to complete. For Ostrich and NPB, we can adjust the benchmark size to reduce the measurement errors. However, this is not the case for Polybench, as the input size is hardcoded. On the other hand, SableWasm performs better than Wasmer in most of the benchmark cases. Here we will take `floyd-warshall` as an example. The core computation function in `floyd-warshall` is a nested for-loop that iteratively multiplies



(a) Polybench

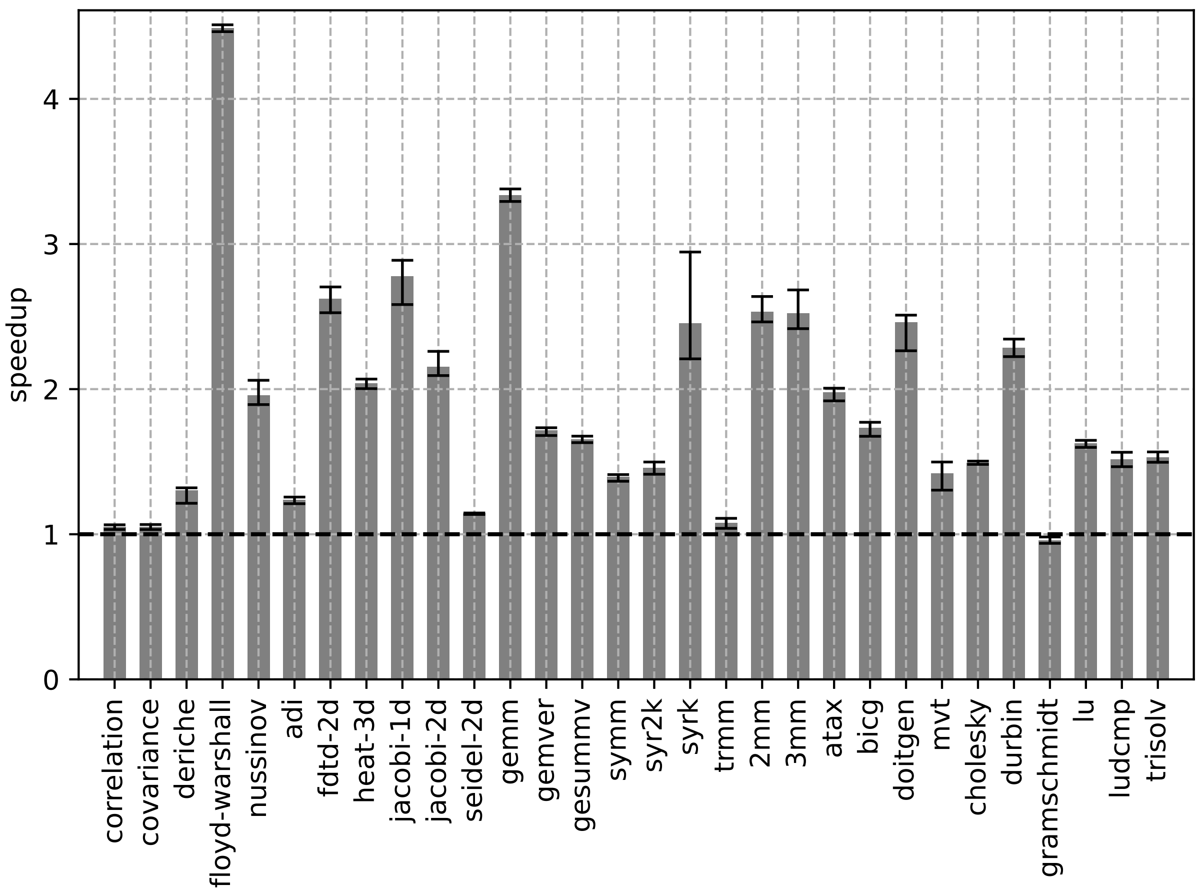


(b) Ostrich

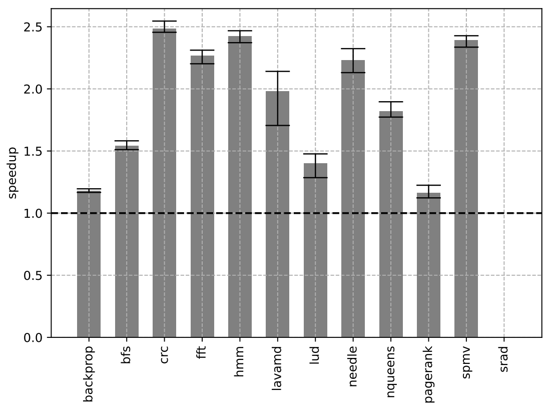


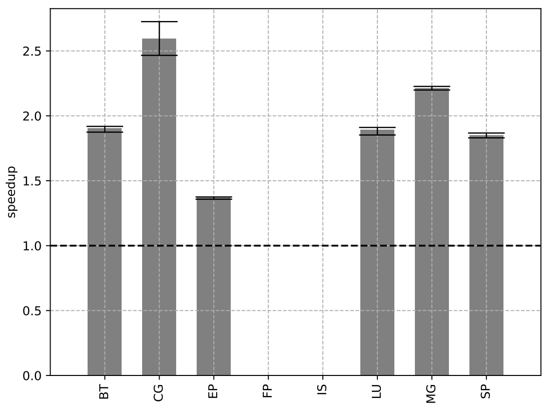
(c) NPB

Figure 7.4: Benchmarks with optimized (`-O3`) on Wasmtime



(a) Polybench

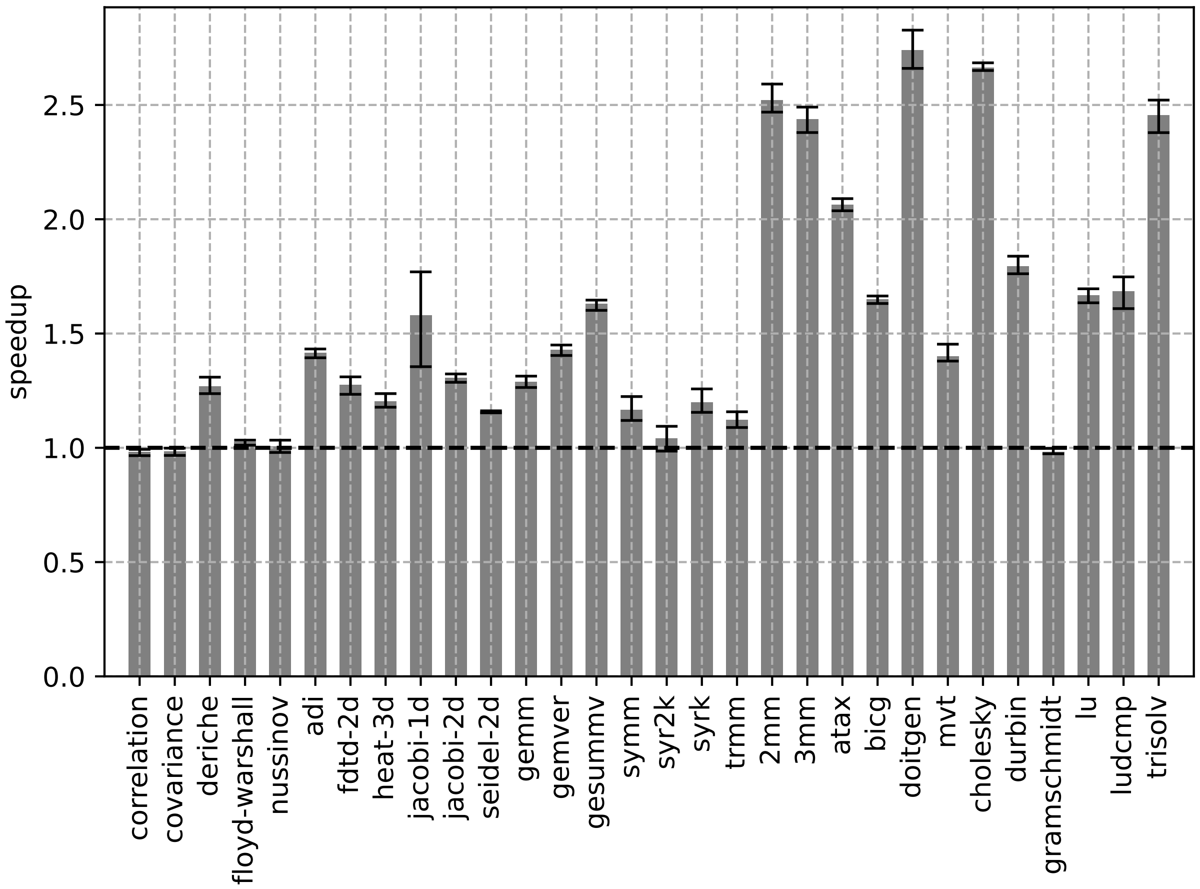




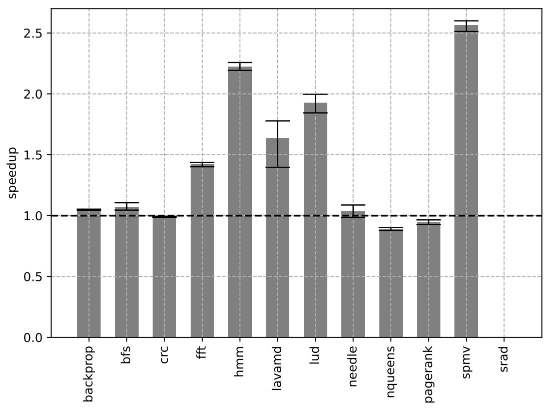
(b) Ostrich

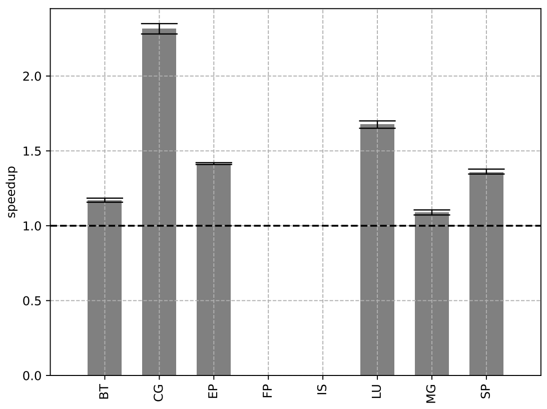
(c) NPB

Figure 7.5: Benchmarks with optimized (`-O3`) on Wasmer (Cranelift)



(a) Polybench

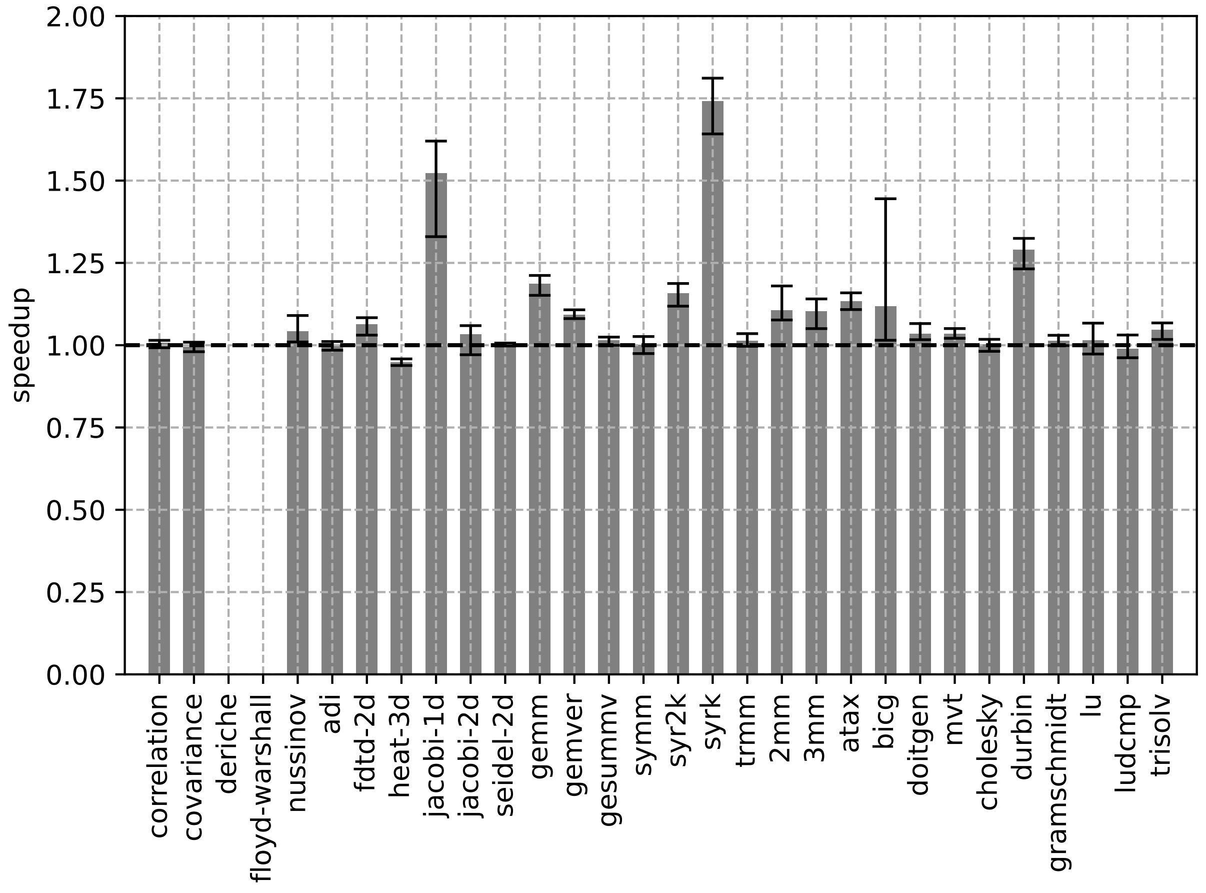




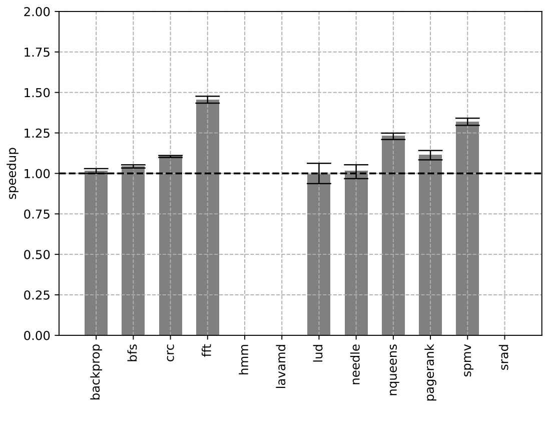
(b) Ostrich

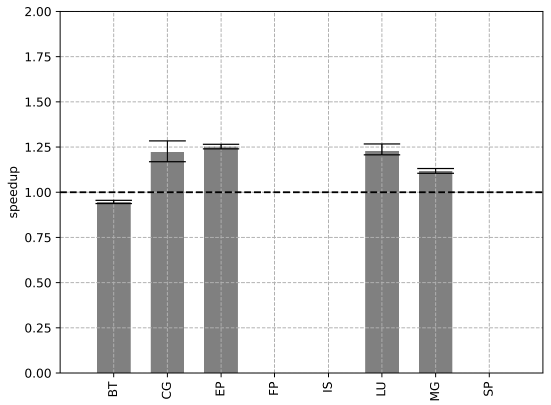
(c) NPB

Figure 7.6: Benchmarks with optimized (`-O3`) on Wasmer (LLVM)



(a) Polybench

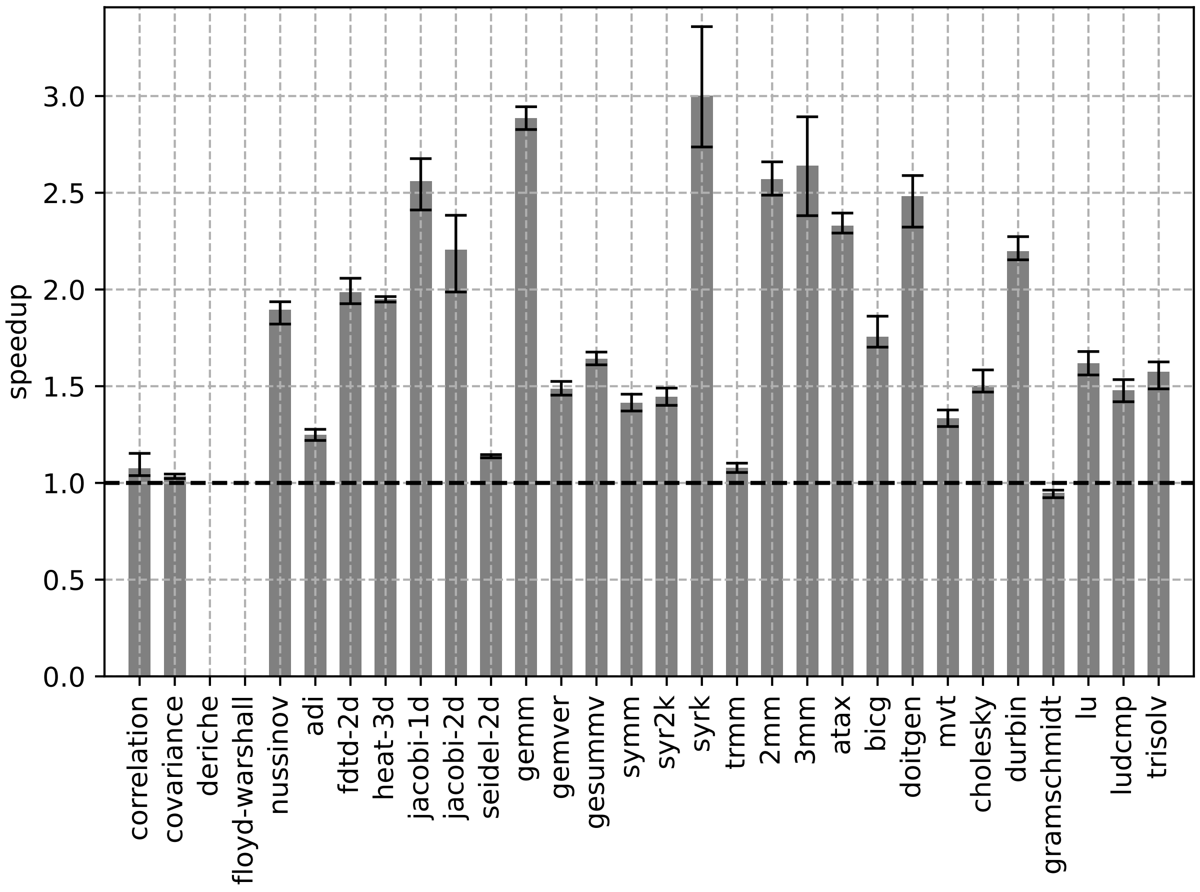




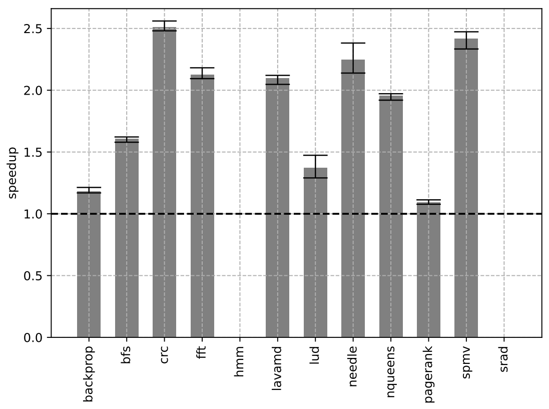
(b) Ostrich

(c) NPB

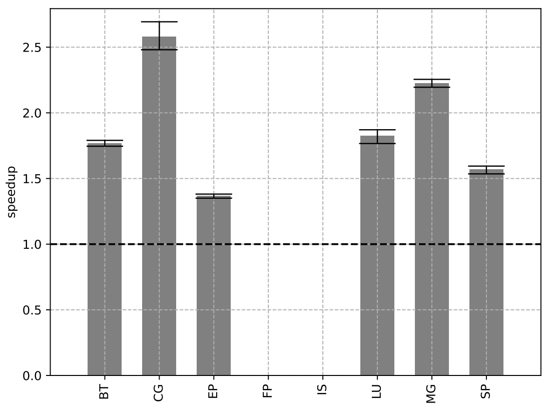
Figure 7.7: Benchmarks with SIMD extension (`-O3 -msimd128`) on Wasmtime



(a) Polybench

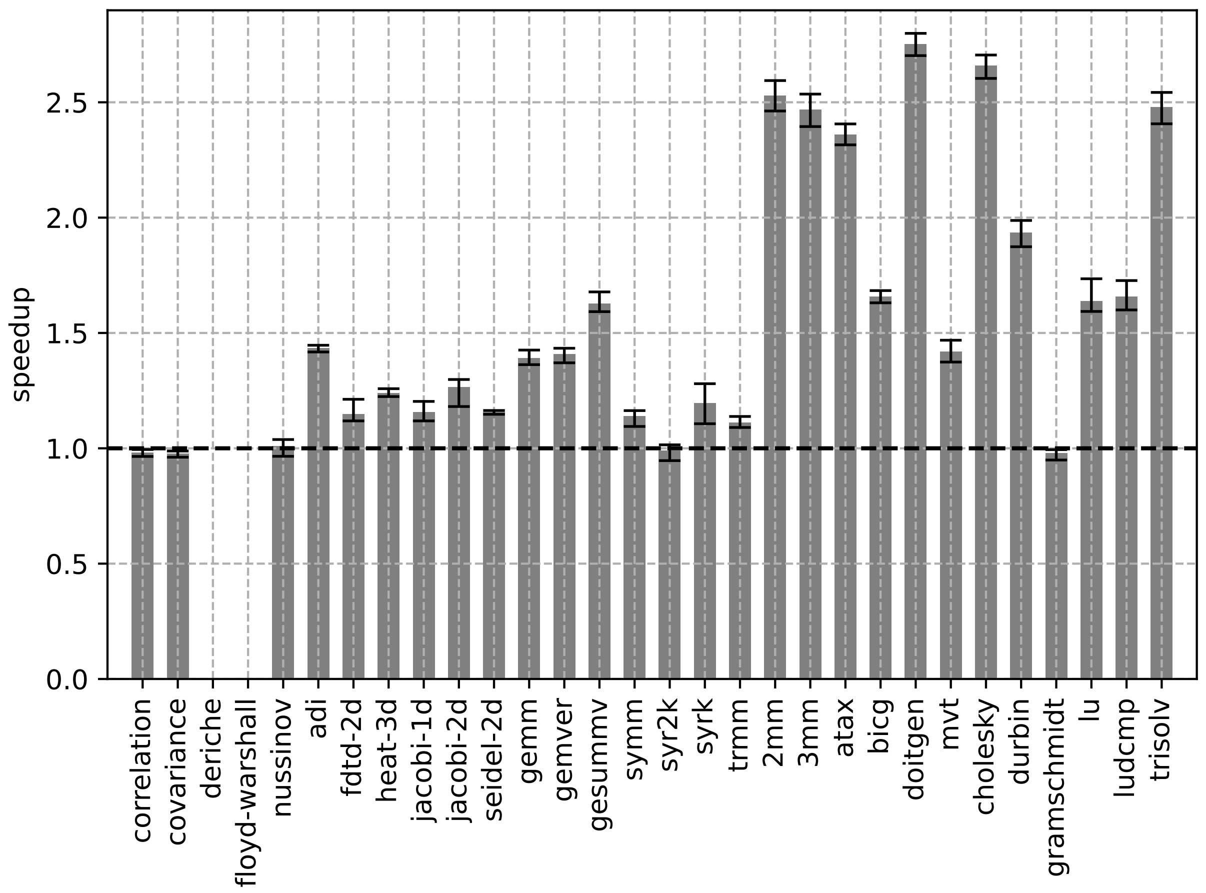


(b) Ostrich

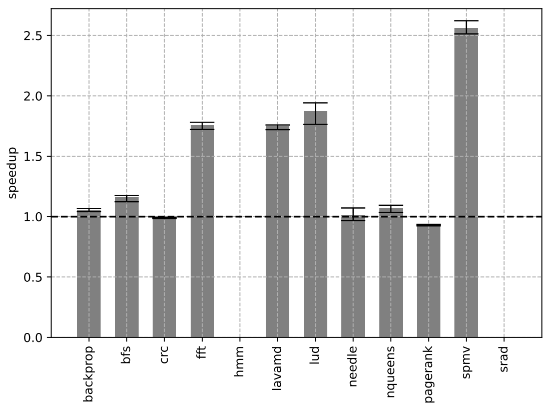


(c) NPB

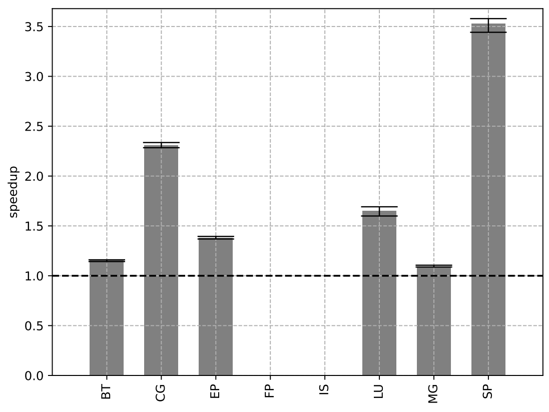
Figure 7.8: Benchmarks with SIMD extension (`-O3 -msimd128`) on Wasmer (Cranelift)



(a) Polybench



(b) Ostrich



(c) NPB

Figure 7.9: Benchmarks with SIMD extension (`-O3 -msimd128`) on Wasmer (LLVM)

Benchmark name		Wasmtime	Wasmer (Cranelift)	Wasmer (LLVM)
Polybench	Naive	1.16	3.04	1.46
	Optimized	1.09	1.77	1.46
	SIMD-enabled	1.09	1.71	1.47
Ostrich	Naive	1.06	2.39	1.32
	Optimized	1.13	1.83	1.34
	SIMD-enabled	1.13	1.79	1.33
NPB	Naive	1.20	3.05	1.79
	Optimized	1.14	1.93	1.45
	SIMD-enabled	1.15	1.85	1.69

Table 7.4: Geometric mean of speedups compare to Wasmtime and Wasmer

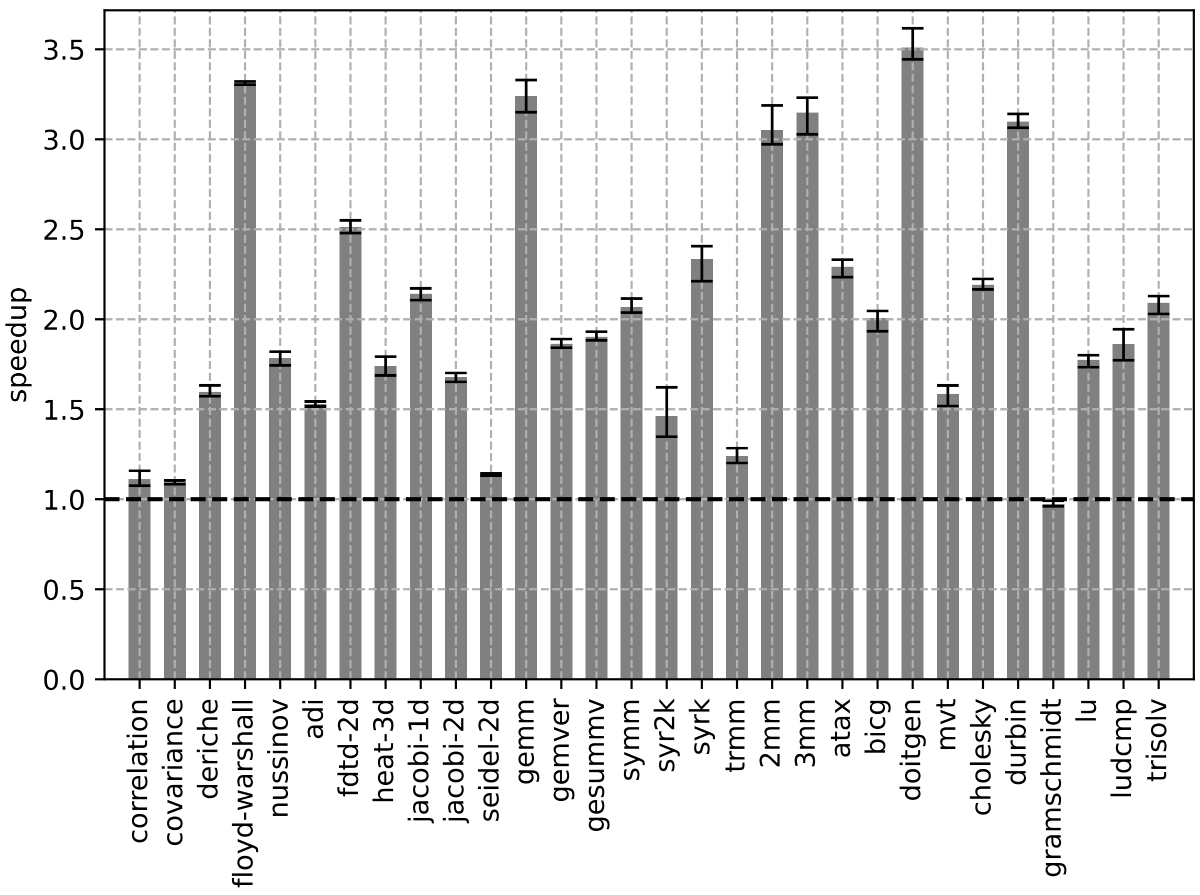
then adds matrices. This operation is highly parallel. We notice that the performance of SableWasm is approximately four times better in optimized input and two times better for SIMD-enabled. Currently, there seems no way to retrieve generated LLVM IR from Wasmer, and we can only speculate on reasons based on the experiment results. We suspect that the auto-vectorization may cause this in the LLVM framework. The four times and two times speedup appears to align with the SIMD vector operations for packed double-precision floating-point numbers. Thus, Wasmer may contain an awkwardly generated LLVM intermediate representation that stops the auto-vectorization pass to turn scalar code into the vectorized form.

In general, we conclude that SableWasm performs on par with Wasmtime on optimized and SIMD-enabled WebAssembly modules and better than Wasmtime and Wasmer for other benchmark cases, as shown in table 7.4.

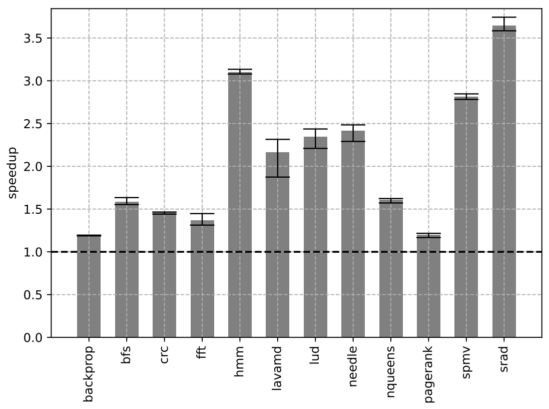
7.3 RQ2: Does optimization over input modules matter?

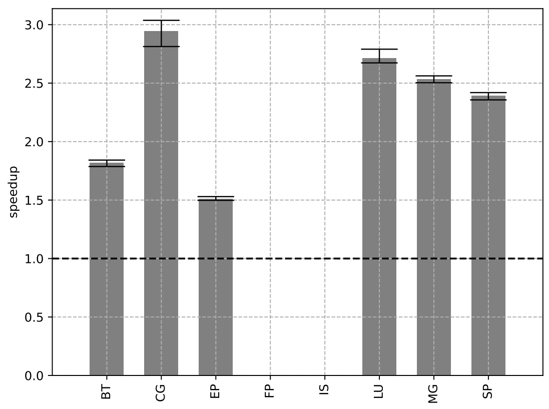
The second problem we would like to investigate in the thesis is whether the optimization over input WebAssembly modules affects their performance in SableWasm. In theory, a perfect runtime system should recover all the missed possible optimization from a naive translated WebAssembly module, which we have seen in many other systems with two-phase compilation. The most well-known example is perhaps Java. The first compiler, `javac`, translates Java source files into bytecodes, and on the other hand, the Java virtual machine (JVM) generates naive executable binaries based on the bytecodes at runtime. `javac` will translate the source files faithfully, without complex transformation and optimization, while the JVM optimizes the bytecodes effectively. This design helps the system to achieve fast compilation while ensuring the quality of the final generated code. In the current SableWasm, however, the optimization over input WebAssembly modules does have a significant impact on the overall performance. When investigating the problem, we compare the SableWasm execution time under naive translated WebAssembly modules against their optimized counterparts.

For most of the benchmarks, except `gramschmidt`, we have seen a significant performance increase for optimized WebAssembly modules, as shown in figure 7.10. Here we will take `trisolv` as an example. We found many optimizations missing when comparing the LLVM intermediate representation generated from a naive WebAssembly module against an optimized one. The most notable difference is perhaps function inlining and loop auto-vectorization. In LLVM generated by naive WebAssembly module, SableWasm does not inline the primary function `kernel_trisolv` into the main function. Thus, matrices are passed as arguments to the computation kernel via pointers, which holds back SableWasm from transforming the internal nested loops into parallel form due to possible data dependencies. We suspect that the translation patterns used in SableWasm when lowering WebAssembly bytecode into SableWasm MIR confuse the LLVM backend. Another interesting question is why `gramschmidt` has similar performance on both input WebAssembly modules. When we compare the LLVM intermediate representation for both inputs, we found that SableWasm can recover nearly all the optimization in the computation kernel: the computation kernel for `gramschmidt` is extremely simple, only consisting of three non-nested for loops. This further confirms our theory on performance difference.



(a) Polybench





(b) Ostrich **(c)** NPB

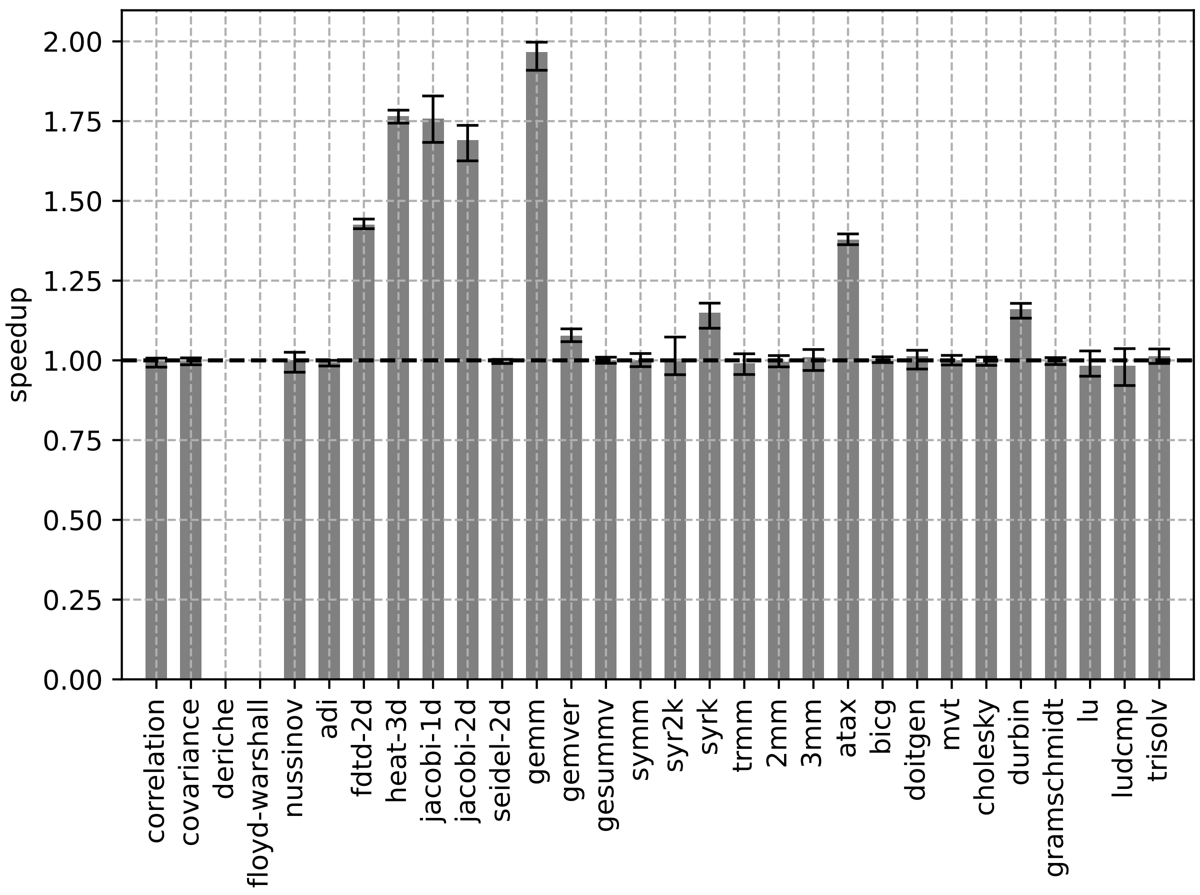
Figure 7.10: Comparision between optimized and naive input modules

7.4 RQ3: How much does SIMD extension improve in performance?

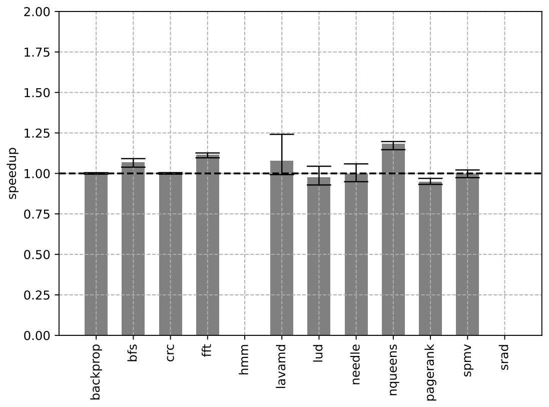
```
void kernel_gemm(int ni, int nj, int nk,
    DATA_TYPE alpha, DATA_TYPE beta,
    DATA_TYPE POLYBENCH_2D(C,NI,NJ,ni,nj),
    DATA_TYPE POLYBENCH_2D(A,NI,NK,ni,nk),
    DATA_TYPE POLYBENCH_2D(B,NK,NJ,nk,nj)) {
    int i, j, k;

    //BLAS PARAMS
    //TRANSA = 'N'
    //TRANSB = 'N'
    // => Form C := alpha*A*B + beta*C,
    //A is NIxNK
    //B is NKxNJ
    //C is NIxNJ
    for (i = 0; i < _PB_NI; i++) {
        for (j = 0; j < _PB_NJ; j++)
            C[i][j] *= beta;
            for (k = 0; k < _PB_NK; k++) {
                for (j = 0; j < _PB_NJ; j++)
                    C[i][j] += alpha * A[i][k] * B[k][j];
             }
        }
    }
}
```

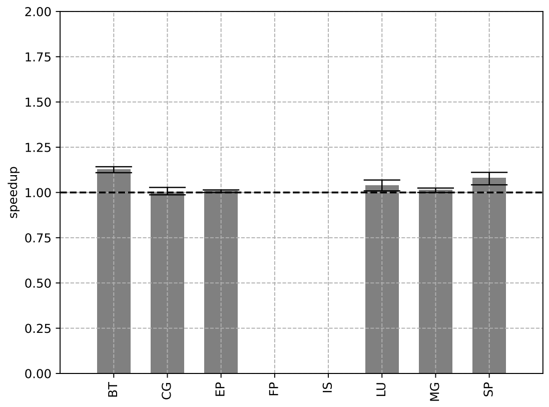
Figure 7.11: Polybench `gemm` benchmark kernel

The last question we would like to investigate in the thesis is how much the WebAssembly SIMD operation extension improves performance compared to MVP WebAssembly. In this experiment, we compare the execution time between optimized WebAssembly modules and SIMD-enabled WebAssembly modules. Figure 7.12 illustrates the experiment results. For most benchmark cases, the SIMD extension does not significantly improve the performance, except for five cases in Polybench. Here we will take `gemm` as an xample. Figure 7.11 illustrates the computation kernel of the benchmark, and it consists of a single nested loop that performs floating-point mathematics over two matrices. When translating the optimized (`-O3`) input WebAssembly module, illustrated in figure 7.13a, SableWasm correctly performs loop unrolling on the inner for-loop. However, the LLVM auto-vectorizer failed to transform the scalar code into a parallel form. On the other hand, in the case of SIMD-enabled WebAssembly module input (`-O3 -msimd128`), SableWasm



(a) Polybench





(b) Ostrich

(c) NPB

Figure 7.12: Comparision between SIMD-enabled and optimized input modules

```
%indvars.iv144 = phi i64 [ 0, %78 ], [ %indvars.iv.next145.1, %79 ]
%.090 = phi i32 [ 1, %78 ], [ %102, %79 ]
%80 = trunc i64 %indvars.iv144 to i32
%81 = add i32 %.096, %80
%82 = urem i32 %.090, 1000
%83 = sitofp i32 %82 to double
%84 = fdiv double %83, 1.000000e+03
%85 = load %__sable_memory_t*, %__sable_memory_t** %12, align 8
%86 = ptrtoint %__sable_memory_t* %85 to i64
%87 = zext i32 %81 to i64
%88 = add nuw i64 %86, %87
%89 = inttoptr i64 %88 to double*
store double %84, double* %89, align 8
%90 = add nuw nsw i32 %.090, %.0100
%91 = trunc i64 %indvars.iv144 to i32
%92 = or i32 %91, 8
%93 = add i32 %.096, %92
%94 = urem i32 %90, 1000
%95 = sitofp i32 %94 to double
%96 = fdiv double %95, 1.000000e+03
%97 = load %__sable_memory_t*, %__sable_memory_t** %12, align 8
%98 = ptrtoint %__sable_memory_t* %97 to i64
%99 = zext i32 %93 to i64
%100 = add nuw i64 %98, %99
%101 = inttoptr i64 %100 to double*
store double %96, double* %101, align 8
```

(a) Optimized

```
%indvars.iv181 = phi i64 [ %indvars.iv.next182, %80 ], [ 0, %78 ]
%81 = phi <2 x i64> [ %102, %80 ], [ <i64 0, i64 1>, %78 ]
%82 = trunc i64 %indvars.iv181 to i32
%83 = add i32 %.0126, %82
%84 = mul <2 x i64> %81, %79
%85 = add <2 x i64> %84, <i64 1, i64 1>
%86 = bitcast <2 x i64> %85 to i128
%87 = and i128 %86, 79228162495817593524129366015
%88 = bitcast i128 %87 to <2 x i64>
%89 = extractelement <2 x i64> %88, i64 0
%90 = urem i64 %89, 1000
%91 = sitofp i64 %90 to double
%.splatinsert1 = insertelement <2 x double> poison, double %91, i32 0
%92 = extractelement <2 x i64> %88, i64 1
%93 = urem i64 %92, 1000
%94 = sitofp i64 %93 to double
%95 = insertelement <2 x double> %.splatinsert1, double %94, i64 1
%96 = fdiv <2 x double> %95, <double 1.000000e+03, double 1.000000e+03>
%97 = load %__sable_memory_t*, %__sable_memory_t** %12, align 8
%98 = ptrtoint %__sable_memory_t* %97 to i64
%99 = zext i32 %83 to i64
%100 = add nuw i64 %98, %99
%101 = inttoptr i64 %100 to <2 x double>*
store <2 x double> %96, <2 x double>* %101, align 8
```

(b) SIMD-enabled

Figure 7.13: Polybench `gemm` code snippet

takes the direct hint from the WASI-enabled clang compiler and correctly emits the 128-bit wide vector operations, as shown figure 7.13b.

Conclusion

In this chapter, we evaluated the performance of SableWasm in terms of three aspects. First, we compared SableWasm's performance against several other well-known WebAssembly runtime environments, Wasmtime and Wasmer. We concluded that SableWasm performs on par with Wasmtime when the input module is optimized and better in all other cases. Second, we investigated whether the optimization over input modules affects the system's overall performance. Currently, SableWasm heavily relies on the frontend compiler to emit efficient code, as the performance gap between naive and optimized input modules is still quite significant. Finally, we evaluated the effectiveness of WebAssembly SIMD operation extensions. Our experiments identified several benchmark cases where explicit SIMD instructions make a measurable difference in the execution speed.

Chapter 8

Related Work

In previous chapters, we presented the SableWasm and evaluated its performance against several well-known benchmark suites. SableWasm is not the only WebAssembly runtime environment system that allows sandboxed WebAssembly modules to run outside of the browser. This chapter will provide a quick overview of several existing WebAssembly host environment implementations. Also, SableWasm does not implement any auto-vectorization algorithms and heavily depends on LLVM, both in the frontend WASI-enabled Clang compiler and the SableWasm backend, to generate parallel code. Auto-vectorization is one of the key research fields in compiler optimization, and much research has been devoted to the field. Therefore, we will briefly cover auto-vectorization in LLVM in this chapter.

WebAssembly runtime environments

In chapter 7, we mentioned two WebAssembly runtime environments developed by the community, Wasmtime and Wasmer. Wasmtime perhaps is the earliest non-browser WebAssembly runtime environment. It started as a side project during WebAssembly standardization and is maintained by Bytecode Alliance[1]. This cross-industry nonprofit or-

[1]Bytecode Alliance: `https://bytecodealliance.org/`

ganization focuses on extending WebAssembly and WASI beyond the browser and IoT devices. Wasmtime is built on the Cranelift compiler framework [2]. Cranelift is similar to LLVM, providing a target-independent intermediate representation that eventually translates to native executable machine code. Currently, at the time of thesis writing, Cranelift is still at very early stages and only supports the x86-64 target. Although the Cranelift started as the backend for Wasmtime, it is not limited to the Wasmtime project. In the future, Cranelift may replace the fast debug backend in the Rust compiler toolchain and the Javascript/WebAssembly engine backend in SpiderMonkey.

Wasmer [3] is another WebAssembly runtime environment and maintained by a startup company. Wasmer shares many similarities comparing to Wasmtime. However, it is more flexible in design. Currently, Wasmer has three different backends, LLVM, Cranelift, and a single-pass compiler for fast code generation. Additionally, comparing to Wasmtime, Wasmer is more aggressive in adding features to WebAssembly. For example, Wasmtime only supports WASI as the system interface API, while Wasmer supports both the WASI and Emscripten specifications. Wasmer also comes with a package manager, called WebAssembly Package Manager (WAPM) [4] which distributed pre-compiled sandboxed WebAssembly binary modules for various applications.

Wasmtime and Wasmer are both just-in-time (JIT) WebAssembly runtime environments. There are also ahead-of-time (AOT) compilers for WebAssembly modules. The most notable one is perhaps the Lucet compiler. Lucet[5] is developed by Fastly and shares a similar design to SableWasm. The initial motivation for Lucet is to create a cloud application system that hosts user-uploaded WebAssembly modules. Currently, Lucet powers Fastly's Terrarium platform, an in-browser multi-language IDE. The Lucet compiler system has two parts, the Lucet shared library compiler, and the Lucet shared library loader. The Lucet shared library compiler compiles WebAssembly modules into shared libraries, while Lucet shared library load dynamically loads the shared library and executes the

[2]Cranelift:`https://github.com/bytecodealliance/cranelift`
[3]Wasmer:`https://wasmer.io/`
[4]WebAssembly Package Manager (WAPM): `https://wapm.io/`
[5]Lucet: `https://www.fastlylabs.com/`

entry function, `_start`. Unlike SableWasm, Lucet is also built on the Cranelift compile framework.

All the WebAssembly environments we have discussed in the section are built on complex compiler frameworks such as Cranelift or LLVM. Therefore, one question that arises naturally is whether WebAssembly is suitable in a resource-constrained environment such as an embedded system. Scheidl shows that it is possible to translate WebAssembly bytecode, under these conditions, into native executable code while maintaining decent performance [29]. One interesting application for WebAssembly is to use it as a form of distributing programs on IoT devices. For example, Jacobsson and Willén implement a WebAssembly interpreter on an SoC that communicates and receives modules from a host device [10]. The system runs on low-power Bluetooth, and in theory, can be used on a wearable device. Another WebAssembly runtime system, Twine, presented in paper [21], focuses on taking advantage of hardware features to further improve the performance of WebAssembly in trusted execution environments (TEE). For example, Twine takes advantage of the Intel SGX instruction set to ensure the module's security and achieve up to 4.1x speedup in performance. This, of course, comes with the drawback of additional hardware-specific dependencies.

Auto-vectorization

This section will briefly discuss auto-vectorization in compilers, more specifically, the LLVM compiler framework. Modern CPU architectures support vector operations to some degree, such as SSE [27], AVX [5] on x86, and Neon [11] on Arm. In recent years, scalable vector extensions, such as Arm's SVE [31], offer even more flexibility on vector size. Although these SIMD instruction set extensions speed up the resulting program, programmers need to have an in-depth understanding of the hardware system to handle them correctly through inlined assembly or intrinsic functions. Additionally, these methods are highly hardware-specific and cause troubles when porting programs to another platform. Another approach to the problem is to ask the compiler to generate vectorized code from traditional scalar code, hence the name auto-vectorization, which is implemented in many modern compiler systems, such as GCC [23] and LLVM [6]. Here we will take the LLVM auto-vectorizer as an example.

The first attempt for auto-vectorization in LLVM is the basic block vectorizer. It works with a single basic block at a time and searches for common patterns. If it finds any optimization opportunity, it will rewrite the basic block into parallel form. One might notice that the basic block vectorizer has no understanding of a loop structure and only perform auto-vectorization if and only if the operations are already unrolled. To address this problem, the second generation of auto-vectorizer is a single block loop vectorizer. The single block loop vectorizer can recognize simple loop structures and consists of two parts the loop legalizer and the loop transformer. The loop legalizer determines if a loop structure can undergo auto-vectorization, and if so, the loop transformer will perform the rewrite. The single block loop auto-vectorizer can also perform loop unrolling if the induction variables are detected. However, this sometimes leads to very aggressive optimization, which slows down the generated code. Hence, in late 2012, the LLVM developers extended the auto-vectorizer with a cost model [24, 25]. The cost model will determine whether a potential optimization worth it based on the instruction set available on the target hardware and data dependency between operands.

LLVM also performs another type of auto-vectorization called superword-level parallelism (SLP) auto-vectorization [14]. SLP auto-vectorization combines similar operations into vector operations, such as memory access and numerical comparison. SLP auto-vectorization is similar to the basic block auto-vectorizer discussed earlier in this section, except that it searches patterns in a bottom-up fashion [7].

Although auto-vectorization brings a silver lining to systemically transforming scalar code into parallel form, it still suffers several drawbacks. The most notable problem is that the dependency between instructions is usually not apparent to the compiler, es-

[6]Auto-vectorization in LLVM: `https://llvm.org/docs/Vectorizers.html`

[7]`https://llvm.org/devmtg/2018-04/slides/Rocha-Look-Ahead%20SLP.pdf`

pecially in a nested loop structure. Hence, the compiler can only take a conservative approach when scheduling the program. One possible solution is to employ a polyhedral model [26] to analyze the data dependency among variables. The polyhedral analysis creates polyhedra based on the program and applies affine transformations to improve instruction scheduling incrementally. LLVM implements the polyhedral analysis in the project Polly [7] [8], which can be used as a compiler plugin and generates scheduling and scope information for instructions. Later, the LLVM loop auto-vectorizer can take advantage of this information to provide better cost estimation. Recent work on auto-vectorization has also exported the use of machine-learning algorithms to make better decisions on cost versus benefit than can be done by simple cost models [32].

[8]LLVM Polly: `https://polly.llvm.org/`

Chapter 9

Future Work and Conclusion

WebAssembly has been growing in popularity in recent years as a new format for distributing sandboxed applications over the internet. In this project, we presented SableWasm as a standalone ahead-of-time (AOT) compiler for translating WebAssembly modules into shared libraries. We also implement a runtime environment that enables other programming languages such as C/C++ to interact with generated shared libraries.

We first started the project with a custom parser for WebAssembly binary format. The parser focuses on extensibility and performance, as currently, WebAssembly is still under the standardization process and several syntax extensions might be merged to the specification soon. We then evaluate the performance of the parser by benchmarking against `wabt`, the reference implementation provided by the WebAssembly community, and observe a 1.6x speedup in execution speed and a 4.6x reduction in memory footprint.

We then define the middle-level representation (MIR) for SableWasm. SableWasm MIR is a register-based control flow graph representation of the WebAssembly program. When translating WebAssembly bytecode to SableWasm MIR, we focus on two significant problems. First, WebAssembly is a stack-based bytecode with structured control flow structures. Thus, to faithfully translate the bytecode, we define translation patterns that mimic the semantics of these constructs and reduce them into basic blocks and branching instructions. The other problem we encountered is regarding the size of the instruction set. As WebAssembly encodes both type and shape information in the instruction opcode, the WebAssembly instruction set is quite large. Additionally, to reduce the size of the module, WebAssembly fuses several typical instruction sequences into one single instruction, such as load-and-extend. To maintain a small instruction set, we define several reduction rules for WebAssembly instructions. Unfortunately, these translation patterns lead to awkward and inefficient code. To address this problem, we implement an analysis and transformation framework over SableWasm MIR. We then design simplifying control flow graph pass that incrementally improves the MIR, similar to a 'peephole optimization' by locating and replacing several common redundant patterns.

The last component of SableWasm is the SableWasm runtime library, which provides implementations for builtin functions used in the generated shared libraries. It also implements several WebAssembly entities, such as the linear memory and the indirect table. Currently, the SableWasm runtime library defines an easy-to-use C/C++ interface to the user and handles errors and exceptions using the exception mechanism in C++.

Finally, we evaluate SableWasm's performance by benchmarking with three well-known benchmark suites, Polybench, Ostrich, and NPB. The first question we focus on in this thesis is how SableWasm performs compared to other existing WebAssembly runtime environments. We conclude that SableWasm performs on par with Wasmtime and approximately 1.5x to 2x faster than Wasmer. The second research question is whether optimization over input WebAssembly modules affects the overall performance in SableWasm. By comparing the execution time of SableWasm under optimized translated input modules against that of naive translated input modules, we conclude that, currently, optimization over the WebAssembly modules has a significant impact on the performance. Hence, when designing frontend compilers that target WebAssembly, one should be careful of translation patterns and perform optimizations as early as possible. The last question we investigated in this thesis is whether the WebAssembly SIMD extension brings performance improvement to SableWasm. Experimentally, we can see significant benefit.

However, using Polybench, we locate many common patterns that cannot be identified and rewritten by LLVM's auto-vectorizer in SableWasm.

Future Work

WebAssembly is a relatively new language, and many of its features are still under the standardization phase. SableWasm only covers several WebAssembly extensions such as the multi-value extension and the SIMD operation extension. One excellent opportunity is to implement more WebAssembly extensions in SableWasm, such as the garbage collection (GC) extension and the exception handling extension. Currently, many high-level languages, such as AssemblyScript, require static linking with a non-trivial runtime library when cross-compiling into WebAssembly. Simulating these features results in notable increases in code size and a slow down in performance.

Another interesting direction is to add more analysis and transformation in SableWasm under the optimization framework. More specifically, one can implement an auto-vectorizer in SableWasm at the MIR level. The evaluation chapter shows that the LLVM's auto-vectorizer cannot recognize many apparent patterns and yields inefficient code. We suspect that the boilerplate code introduced by the translation patterns confuses the auto-vectorizer. Hence, an auto-vectorizer at the SableWasm MIR level can better understand the program and, in theory, recover more opportunities within the WebAssembly modules.

Finally, one can also add more backend support for SableWasm. Currently, SableWasm is an ahead-of-time (AOT) compiler built on the LLVM compiler infrastructure. One natural extension of this project to implement a just-in-time (JIT) system that uses LLVM's Orc JIT framework. Additionally, one can also explore many profile-guided optimizations (PGO) techniques used in many other VM languages, such as Java bytecode [1].

Bibliography

[1] Arnold, M., Hind, M., and Ryder, B. G. Online feedback-directed optimization of Java. *ACM SIGPLAN Notices 37*, 11 (2002), 111–129.

[2] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct. 1991), 451–490.

[3] Doherty, J., and Hendren, L. McSAF: A static analysis framework for MATLAB. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Berlin, Heidelberg, 2012), ECOOP'12, Springer-Verlag, p. 132–155.

[4] Fetzer, C. Student forum. In *International Conference on Dependable Systems and Networks (DSN'06)* (2006), pp. 594–594.

[5] Firasta, N., Buxton, M., Jinbo, P., Nasri, K., and Kuo, S. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel white paper 19*, 20 (2008).

[6] Georgiadis, L., Tarjan, R. E., and Werneck, R. F. Finding dominators in practice. *Journal of Graph Algorithms and Applications 10*, 1 (2006), 69–94.

[7] Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Grösslinger, A., and Pouchet, L.-N. Polly-polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)* (2011), vol. 2011, p. 1.

[8] HAAS, A., ROSSBERG, A., SCHUFF, D. L., TITZER, B. L., HOLMAN, M., GOHMAN, D., WAGNER, L., ZAKAI, A., AND BASTIEN, J. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, Association for Computing Machinery, p. 185–200.

[9] HERRERA, D., CHEN, H., LAVOIE, E., AND HENDREN, L. Numerical computing on the web: Benchmarking for the future. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages* (2018), pp. 88–100.

[10] JACOBSSON, M., AND WILLÉN, J. Virtual machine execution for wearables based on webassembly. In *EAI International Conference on Body Area Networks* (2018), Springer, pp. 381–389.

[11] JANG, M., KIM, K., AND KIM, K. The performance analysis of ARM NEON technology for mobile platforms. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation* (New York, NY, USA, 2011), RACS '11, Association for Computing Machinery, p. 104–106.

[12] JANGDA, A., POWERS, B., BERGER, E. D., AND GUHA, A. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 107–120.

[13] LAM, S. K., PITROU, A., AND SEIBERT, S. Numba: A LLVM-based python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (New York, NY, USA, 2015), LLVM '15, Association for Computing Machinery.

[14] LARSEN, S., AND AMARASINGHE, S. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2000), PLDI '00, Association for Computing Machinery, p. 145–156.

[15] LATTNER, C. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[16] LEE, J., HUR, C., AND LOPES, N. P. Aliveinlean: A verified LLVM peephole optimization verifier. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II* (2019), I. Dillig and S. Tasiran, Eds., vol. 11562 of *Lecture Notes in Computer Science*, Springer, pp. 445–455.

[17] LENGAUER, T., AND TARJAN, R. E. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 1 (Jan. 1979), 121–141.

[18] LOPES, N. P., MENENDEZ, D., NAGARAKATTE, S., AND REGEHR, J. Provably correct peephole optimizations with alive. *SIGPLAN Not. 50*, 6 (June 2015), 22–32.

[19] MCKEEMAN, W. M. Peephole optimization. *Commun. ACM 8*, 7 (July 1965), 443–444.

[20] MUSCH, M., WRESSNEGGER, C., JOHNS, M., AND RIECK, K. New kid on the web: A study on the prevalence of WebAssembly in the wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (Cham, 2019), R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds., Springer International Publishing, pp. 23–42.

[21] MÉNÉTREY, J., PASIN, M., FELBER, P., AND SCHIAVONI, V. Twine: An embedded trusted runtime for webassembly, 2021, 2103.15860.

[22] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, Association for Computing Machinery, p. 89–100.

[23] NUZMAN, D., AND HENDERSON, R. Multi-platform auto-vectorization. In *International Symposium on Code Generation and Optimization (CGO'06)* (2006), pp. 11 pp.–294.

[24] POHL, A., COSENZA, B., AND JUURLINK, B. Correlating cost with performance in LLVM. *Proceedings of the 13th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)* (2017).

[25] POHL, A., COSENZA, B., AND JUURLINK, B. Vectorization cost modeling for NEON, AVX and SVE. *Performance Evaluation 140* (2020), 102106.

[26] QUILLERÉ, F., AND RAJOPADHYE, S. Optimizing memory usage in the polyhedral model. *ACM Trans. Program. Lang. Syst. 22*, 5 (Sept. 2000), 773–815.

[27] RAMAN, S. K., PENTKOVSKI, V., AND KESHAVA, J. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE Micro 20*, 4 (2000), 47–57.

[28] SALIM, S. S., NISBET, A., AND LUJÁN, M. TruffleWasm: A webassembly interpreter on GraalVM. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2020), VEE '20, Association for Computing Machinery, p. 88–100.

[29] SCHEIDL, F. Valent-blocks: Scalable high-performance compilation of WebAssembly bytecode for embedded systems. In *2020 International Conference on Computing, Electronics Communications Engineering (iCCECE)* (2020), pp. 119–124.

[30] SHI, Y., CASEY, K., ERTL, M. A., AND GREGG, D. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim. 4*, 4 (Jan. 2008).

[31] STEPHENS, N., BILES, S., BOETTCHER, M., EAPEN, J., EYOLE, M., GABRIELLI, G., HORSNELL, M., MAGKLIS, G., MARTINEZ, A., PREMILLIEU, N., AND ET AL. The ARM scalable vector extension. *IEEE Micro 37*, 2 (Mar 2017), 26–39.

[32] STOCK, K., POUCHET, L.-N., AND SADAYAPPAN, P. Using machine learning to improve automatic vectorization. *ACM Trans. Archit. Code Optim. 8*, 4 (Jan. 2012).

[33] VAN DER WIJNGAART, R. F., AND WONG, P. NAS parallel benchmarks version 2.4. Tech. rep., NASA Advanced Supercomputing (NAS), 2002.

[34] Wagner, L. Turbocharging the web. *IEEE Spectrum 54*, 12 (2017), 48–53.

[35] Watt, C. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New York, NY, USA, 2018), CPP 2018, Association for Computing Machinery, p. 53–65.

[36] Yuki, T. Understanding polybench/c 3.2 kernels. In *International workshop on Polyhedral Compilation Techniques (IMPACT)* (2014), pp. 1–5.

[37] Zakai, A. Emscripten: An LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (New York, NY, USA, 2011), OOPSLA '11, Association for Computing Machinery, p. 301–312.