Julia in WebAssembly

by

Raymond Minor Huffman

B.S. in Computer Science and Engineering
Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

Author .....
Department of Electrical Engineering and Computer Science
January 20, 2023

Certified by .....
Alan Edelman
Professor of Applied Mathematics
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Julia in WebAssembly

by

Raymond Minor Huffman

Submitted to the Department of Electrical Engineering and Computer Science
on January 20, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

WebAssembly is a modern binary instruction format that enables highly performant program execution in sandboxed execution environments. WebAssembly modules can be run natively in web browsers, or within lightweight, isolated, server-side runtimes.

This project explores applications of WebAssembly using the Julia programming language and how languages that leverage just-in-time compilation can function under the particular architectural limitations of WebAssembly. It demonstrates the feasibility of multiple strategies for compiling Julia code to WebAssembly, and details the future work required to run the Julia compiler entirely in WebAssembly.

Thesis Supervisor: Alan Edelman
Title: Professor of Applied Mathematics

Acknowledgments

Thank you to Valentin, who provided endless counseling and support throughout this project, especially when debugging near endless compiler idiosyncrasies.

Thank you to Professor Edelman, who supervised this thesis, for connecting me with Valentin and the Julia Lab, and for enabling me to work on such an interesting and rewarding project.

Lastly, thank you to my family for supporting me during my research, and for always believing that I would, in fact, finish my thesis.

Contents

List of Figures

Chapter 1

Introduction

Web browsers can execute programs written in JavaScript. JavaScript is a very flexible programming language and has become ubiquitous in today's internet for writing user interfaces and web applications. However, JavaScript was not designed for high performance and data analytics applications and suffers in comparison to languages such as C++ and Julia. The former is a lower-level language ideal for writing high performance applications, and the latter is a higher-level language ideal for data analytics, with a growing library of packages addressing the needs of wide variety of use cases.

WebAssembly (Wasm) was first introduced by Mozilla in 2017, and enables programs written in any supported language to be compiled into modules and run in a web browser. Wasm is a binary instruction format and a compilation target for programming languages. Just as a program can be compiled for x86 or for ARM, it can be compiled for Wasm. Wasm programs can execute at native speed, making them significantly faster than equivalent programs written in JavaScript. Additionally, by acting as a common compiler target for any programming language, a programmer can choose the language and packages that best fit their needs.

A compiler is a program that converts a program written in a human-readable language into a binary executable, such as Wasm, that can be run by a computer. The LLVM project is a collection of modular components that is used to implement compilers for a variety of programming languages. LLVM is built around the LLVM

intermediate representation (LLVM IR). Programs are first converted from a language like C++ to the LLVM IR, and then converted from LLVM IR to an executable binary, like Wasm. Julia implements its compiler using LLVM, which means Julia code can already be converted to LLVM IR. Independently, a Wasm backend has been developed for LLVM, meaning that LLVM IR code can be converted to a Wasm binary.

Chapter 2

Background

2.1 WebAssembly

Use cases for WebAssembly can be divided into two categories: web and non-web.

For the web environment, Wasm support for Julia would enable applications written in Julia to run natively in a web browser without the need for users to download and install a local copy of Julia. As an example, consider interactive notebooks like Jupyter and Pluto.jl. To view and interact with these notebooks, users must install a local copy of Julia, install all the notebook's required packages, and have some ability to debug any issues related to this installation process. An alternative is a cloud-based product like Google CoLab, which provides a backend on their servers for a fee. Wasm support for Julia would provide a third option: users could open Julia notebook directly in their web browser without the need to install any software. This would also expand support to platforms like iOS, Android, and ChromeOS that do not have native Julia support.

In addition to interactive notebooks, any web application that would currently offload complex computation to a backend server could instead implement that logic in Julia and execute it locally on the user's device. Applications such as image processing, generating charts and figures, and solving differential equations are just a few examples.

For the non-web environment, WebAssembly acts as a flexible portable binary

format for containerized compute. WebAssembly containers offer the same isolation guarantees as virtual machines but are significantly more lightweight. Wasm modules have no direct access to memory, filesystem, network, or other unsafe system APIs. Additionally, safe memory access and control flow integrity are guaranteed. Wasm uses the Harvard architecture, strictly segregating instruction and data memory, making it impossible to modify the call stack with buffer overflow attacks. Control flow integrity is guaranteed by Wasm's execution semantics without the need for additional runtime instrumentation. These numerous security guarantees make Wasm a prime candidate for safely executing untrusted code.

2.2 Julia

Julia is a dynamically typed just-in-time compiled language. To understand what this means, consider a statically typed ahead-of-time (AOT) language like C. In C, the programmer declares the concrete data types of all variables, each with specific low-level representations in memory. In order to execute a C program, it is compiled to produce a platform-specific executable. The fact all variables have specific programmer-defined types is called "static typing," and the requirement to compile a program before it is executed is called "ahead-of-time" or "static" compilation.

Now, consider a language with dynamic typing, such as Python. In Python, programmers do not declare any types for variables. However, in order to run any program, a computer must know the concrete types of all variables in order to properly represent their data in memory. For example, integers and decimal variables for the same value have completely different physical representations in memory. By the time a program is converted to machine code and executed, variables must have concrete data types defined. Python supports dynamic typing because it is an interpreted language, which means that programs are converted line-by-line into executable machine code as the program is run. As each line is passed to the interpreter, the types of each variable are checked, and the correct machine code for each operation is executed with concrete types defined. However, this type checking at every line

introduces significant overhead and is one of the reasons why Python programs are significantly slower than equivalent Julia programs.

Julia is also an interpreted language, it but leverages a Just-In-Time (JIT) compiler to improve performance. In Julia, as in Python, functions are defined using arguments with unspecified data types. At runtime, the first time a function is called, the Julia interpreter determines the concrete types of each argument. Then, it passes the function definition along with the inferred argument data types to the compiler which compiles a bespoke implementation of the function with those particular types. This compiled version of the function is cached so that subsequent calls with the same types of arguments simply execute this machine code as if it had been compiled ahead of time, reaping all the performance benefits of AOT compilation.

The Julia JIT compiler relies on the fact that most modern processors use the Von Neumann architecture. In this model of processor design, program instructions and program data are physically stored on the same memory bus and are essentially interchangeable. This means that data created by a program can be marked as instructions and executed as if it were additional lines of machine code. This is precisely how a JIT compiler works. The compiler is passed a block of Julia code with concrete type information, and machine code is generated and stored as regular data in memory. The machine code is then marked as executable instruction memory, and a pointer to the first step in the function is stored as a function pointer where it can later be looked up and called.

WebAssembly instead uses the Harvard architecture, meaning that instructions and data are strictly segregated, and instruction memory is locked as read-only. This means that while it would be possible to generate Wasm machine code at runtime as normal, that code could not be immediately marked as executable and run - it can only be output to a separate binary file.

To circumvent this limitation, Wasm's support for dynamic linking must be leveraged. Dynamic linking is primarily used to load the dependencies of programs, called libraries. When a library is dynamically linked at runtime, the program opens the library file by calling `dlopen()` and loads the functions it requires by calling `dlsym()`.

Linking usually occurs immediately after compilation or during a program's initialization phase, but linking using `dlopen` and `dlsym` can occur at any time. In order to function in WebAssembly, the Julia JIT compiler could compile a function to Wasm at runtime, save the output as a library file, and then call `dlopen` and `dlsym` to load the machine code and make it executable.

The challenge here is not the concept of implementing a JIT compiler in Wasm but in the implementation details of the Julia compiler that make retrofitting this dynamic linking behavior particularly difficult.

Chapter 3

Related Work

Emscripten

Emscripten is a compiler toolchain for Wasm. It provides a suite of tools for compiling C/C++ programs to Wasm. While it is possible to use clang and LLVM to compile for Wasm, one is limited to compiling simple programs that do not make use of system calls or the C standard library (libc). Emscripten solves this problem by emulating a POSIX operating system, which implements the system calls required to use libraries like libc. Emscripten also implements fundamental memory operations like malloc.

Using Emscripten in place of a standard C/C++ compiler is straightforward, as its executable, `emcc`, functions as a drop-in replacement for `gcc` or `clang`.

Non-web Runtimes

While WebAssembly was designed primarily to run in web browsers, Wasm binaries can also be run in non-web containers. Wasmtime and Wasmer are two competing implementations of non-web runtime environments for Wasm.

WASI

The WebAssembly System Interface (WASI) is an API for interacting with OS features like filesystems and network sockets when running Wasm in a non-web environment.

Pyodide

Pyodide is a version of Python and a collection of scientific computing packages compiled to Wasm. This project has successfully packaged the Python interpreter in a Wasm module, enabling Python code to be authored and executed entirely in a web browser. Another relevant project is JupyterLite, which uses Pyodide to implement a Python interactive notebook running entirely in Wasm.

Pluto.jl

Pluto is a reactive interactive notebook for Julia. Unlike Jupyter notebooks, which require users to manually run and re-run cells as code is changed, Pluto automatically responds to code changes by updating all affected cells. A Pluto notebook running entirely in Wasm is an exciting application of this thesis work.

Chapter 4

Work Done

4.1 Static Compilation of Julia Programs

The Julia compiler internally uses the LLVM compiller infrastructure to generate native executables. Julia codegen is responsible for converting Julia code to LLVM IR, which is then passed to LLVM to be optimized and translated to platform-specific assembly code. While Julia code is usually just-in-time (JIT) compiled at runtime, the `GPUCompiler.jl` package can be used to compile programs ahead of time, a technique known as static compilation.

`GPUCompiler.jl` is a package used internally by `CUDA.jl` and `AMDGPU.jl` to provide an interface for Julia programs to interact with graphics processing units (GPUs). `GPUCompiler.jl` exposes some internal compiler infrastructure that can be exploited to statically compile Julia programs.

To statically compile a Julia function as a WebAssembly module, we first generate a method instance for the function, specifying concrete types for any arguments. Julia is a dynamically typed language, meaning that concrete types for function arguments are not know until the function is called at runtime. However, when statically compiling, we must specify concrete types as Wasm does not support dynamic typing. To generate a WebAssembly module that supports multiple runtime types, we could generate a method instance for each combination of argument concrete types that we would like to support.

Once method instances are generated, they are converted to LLVM IR and then to an assembly language of our choice. For Wasm, we specify the LLVM triple `wasm32-unknown-unknown`. This tells the LLVM compiler to use its WebAssembly backend implementation to generate a Wasm binary, which is then output to a file. This file can then be executed in a web browser, on in a non-web WebAssembly runtime.

This experiment demonstrates that Julia programs can be statically compiled to WebAssembly. However, as mentioned above, Julia is a dynamically typed JIT compiled language, so static compilation alone is not sufficient for true Julia WebAssembly support, but it is a crucial first step that demonstrates the feasibility of this work.

4.2 Spark WebAssembly UDFs

Apache Spark is a popular system for data processing and analytics. Spark typically runs on a cluster of machines and enables users to process datasets that may be too large to fit on a single machine. Spark provides a simple interface for writing SQL-like queries, but also supports arbitrary user-defined functions (UDFs). UDFs are treated as black boxes for functional operations like map, filter, and reduce. Spark has official support for Java, Scala, Python, and R, and UDFs can be implemented in any of those languages.

Consider a cloud computing provider that hosts shared Spark clusters and allows its users to submit jobs. Spark UDFs would essentially allow users to run arbitrary code on the Spark cluster, with the potential to interfere with and access the filesystems of peer jobs on the cluster. Spark does provide some security features, such as authenticated and encrypted RPCs between processes, authentication for requests to the cluster, and encryption for local temporary files. These features, however, are not sufficient to isolate a malicious job with submit permissions on the cluster. Currently, the best way to isolate Spark jobs it to create separate, isolated clusters, which introduces significant overhead and startup time.

As part of this thesis work, a solution for isolating untrusted Spark UDFs was

```
object SparkUdfRunner {
  def main(args: Array[String]): Unit = {

    val df = readFromCSV(INPUT_PATH)
    val wasmBytes = Files.readAllBytes(WASM_PATH)

    import spark.implicits._
    val outputDf = df.mapPartitions(iterator => {

      // Instantiate Wasm Module
      val instance = new Wasmer.Instance(wasmBytes)
      val execFunction = instance.exports.getFunction("exec")

      val results = new IntBuffer(PARTITION_SIZE)

      for ( i <- 0 to PARTITION_SIZE)
        results[i] = execFunction.apply(
          iterator.next().asInstanceOf[Object]
        )
      }

      results
    })(RowEncoder(SCHEMA))

    outputDf.write.format("csv").save(OUTPUT_PATH)
  }
}
```

Figure 4-1: Spark job initialization pseudocode in Scala

developed using WebAssembly. Specifically, a generic UDF wrapper was implemented in Scala that serves as an interface between Spark Datasets and a WebAssembly runtime. This allows a user to implement arbitrary logic in any language that supports compilation to Wasm and execute that logic on a Spark cluster. Since there is no native support for Julia by the Spark maintainers, this introduces some level of Spark support for Julia.

Figure 4-1 displays the pseudocode for initializing a Spark job and submitting a Wasm module that contains the implementation of a map function. The full implementation can be found in the appendix. The Wasm module is first loaded from disk

serialized so that it may be passed to the Spark workers. Next, a Spark session is initialized, and the input data is loaded and partitioned to each of the worker nodes. We use Spark's 'mapPartitions' API to provide Spark with additional initialization code to run on each worker node before processing each partition. When processing each partition, we deserialize the Wasm code and initialize the Wasm runtime. Then, for each row in the partition, we call the Wasm module and return its output.

To improve performance, an optimization was implemented to reduce the number of calls from Scala to Wasm. While Wasm has a highly restricted memory model and cannot access the JVM memory to access Spark's data, it is possible for the Scala runtime write to and read from the Wasm module's memory. In addition, by exporting `malloc`, we can ask the Wasm module to allocate space in its memory for input and output data, and it will return us pointers to blocks of memory. This means that instead of calling the Wasm module for each row, the entire partition is copied to the Wasm module as arrays and the Wasm module is called with pointers to the input and output arrays. The output data is then copied back to Spark and returned as before.

4.3 Building Julia Targeting WebAssembly

4.3.1 Julia Compiler Architecture

The Julia compiler is fundamentally just a program written in C/C++ that parses Julia code, compiles it to a native binary format using LLVM, and executes it. Julia is just-in-time compiled, which means there is some runtime component that orchestrates the compilation of methods as they are called, loads the newly compiled binary, and executes it. This orchestration layer is called the Julia runtime, and the component that runs the compilation is the Julia codegen.

The core of the Julia source code is divided into two separate libraries: `libjulia-internal` and `libjulia-codegen`, both written in C/C++. A third library, `libjulia`, implements the Julia loader and acts as a wrapper around internal and codegen. It loads

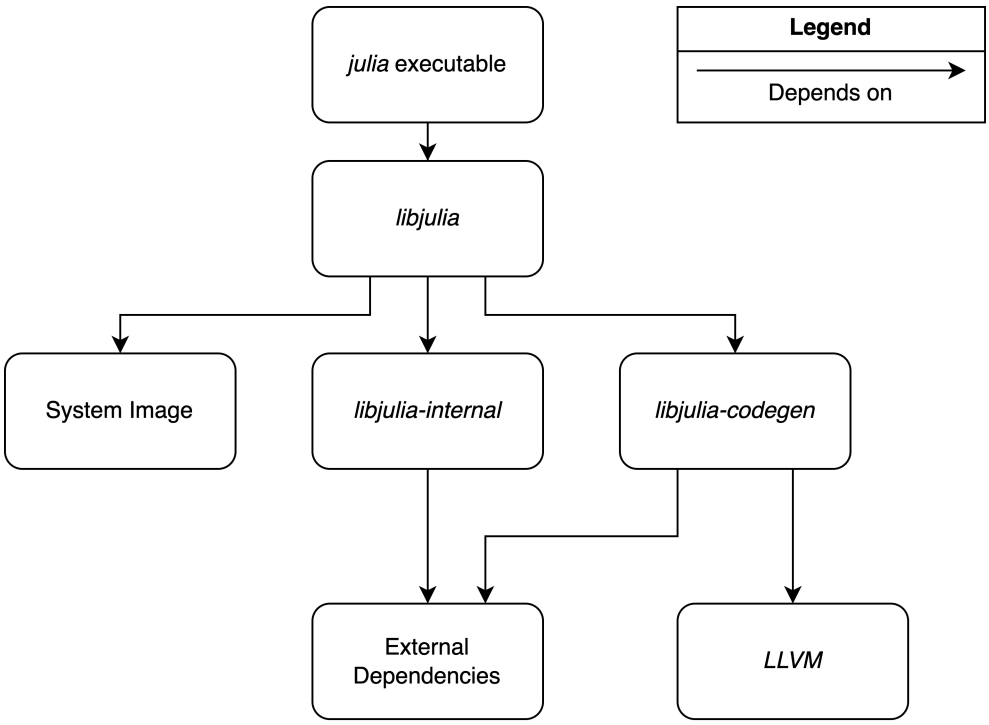

Figure 4-2: Julia compiler architecture

the two libraries at runtime and re-exports their exported symbols as part of a public API, and it is also written in C. The final component is the system image. The sysimg is created by compiling the Julia Base library, which contains implementations of core language functionality written in Julia such as the definitions of basic numeric types, arrays, and arithmetic operations. The sysimg is also a library loaded by `libjulia`. Figure 4-2 is a graphical representation of these library dependencies.

The Emscripten compiler toolchain enables us to build C programs to WebAssembly, so fundamentally, building Julia should be as simple as specifying `wasm` as the target architecture. Unfortunately, several critical components in the compiler are platform-specific, meaning that each architecture has its own implementation of parts of the runtime. In addition, Julia has multiple dependencies that also must be compiled to Wasm, each with their own set of platform specific implementations and other complexities that prevent Emscripten from building them as Wasm libraries with no modifications.

4.3.2 Linking in WebAssembly

When building programs with dependencies, those dependencies, known as libraries, must be linked together to produce a final executable that can be run by a computer. Each library contains a list of exported symbols, which are the names and memory addresses of functions and global variables that are defined in the library. Other code that depends on a library and references those exported symbols imports those symbols by linking. Linkable libraries come in two flavors: static and dynamic. Static libraries are linked at compile time, meaning that the final executable has the binaries for all its dependencies contained within itself. This is very useful for distributing executables that don't require users to install additional dependencies, but it produces much larger executables since all the dependencies are bundled together.

The alternative is to use shared libraries with dynamic linking. Shared libraries are usually standard dependencies that are commonly required, such as math or graphics libraries. Since these libraries are often large, it's inefficient to distribute a separate copy of each library for each program that uses it as a dependency. Instead, executables are built without including all dependencies, and it is assumed that the shared library files will be available to link at runtime. The Julia compiler uses shared libraries with dynamic linking for all its dependencies.

However, the dynamic linking model relies on a few assumptions that, while true on native execution environments, do not hold for WebAssembly. On a standard platform like Windows or macOS, the application runs natively on the local system with access to all system libraries. In addition, the size of these libraries is not critical, as there is only one copy of each library on the system which don't need to be reinstalled each time they are used by a program. In contrast, WebAssembly modules are executed in an isolated container with no access to system libraries, meaning that all dependencies must be bundled with the Wasm module. The size of these libraries is also very important, and they will be freshly downloaded each time a user loads the webpage hosting a Wasm module. It is for these reasons that Emscripten recommends that dynamic linking be avoided when possible.

In most cases, shared libraries can be easily substituted with static libraries. Unfortunately, this is not the case for `libjulia-internal` and `libjulia-codegen`, which are loaded by `libjulia` in a non-standard way. The design of the `libjulia` loader is quite unique and was built specifically to support flexibility with how system libraries are used by Julia, and to support compatibility with different execution platforms that each handle dynamic linking slightly differently. If instead static linking is used, the complexities of `libjulia` become unnecessary. When statically linking for Wasm, `libjulia` is excluded entirely, and the final executable is linked statically with `libjulia-internal` and `libjulia-codegen`.

4.3.3 Building the Julia Runtime

While Julia is primarily just-in-time compiled, the Julia runtime can use its interpreter to execute lines of Julia code without compiling using `libjulia-codegen`. This is useful for simple operations where the additional overhead of compiling a particular line of code would cause it to execute more slowly than if it were interpreted. If, for some reason, Julia is not able to compile a particular function or line of code, it falls back to evaluating using the interpreter. We can exploit this fallback functionality of the compiler to build a version of the Julia runtime that does not include any codegen components.

To build the Julia runtime without codegen, we simply compile `libjulia-internal` and skip the build step that compiles `libjulia-codegen`, not linking it to the final executable. By skipping the build of `libjulia-codegen` entirely, we reduce the number of source files and codegen-specific dependencies that may need patches for Wasm compatibility which should result in a more straightforward build process. This simplified architecture is pictured in figure 4-3.

Building Dependencies

Several dependencies are still required by `libjulia-internal`, each necessitating modifications to build scripts and source code to be built as static Wasm libraries.

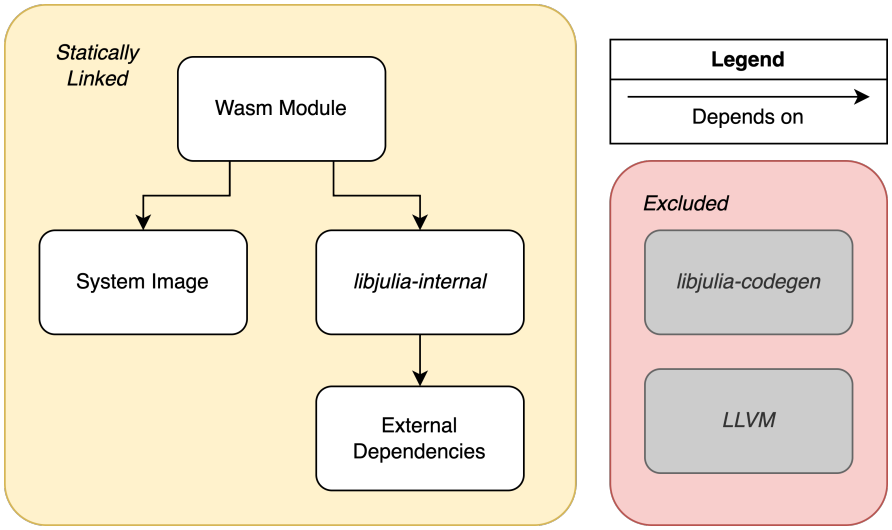

Figure 4-3: Architecture of Julia interpreter-only Wasm module

The dependencies that were modified as part of this work were, `LibUV`, `UTF8proc`, `pcre`, `gmp`, `mpfr`, and `dsfmt`.

For each library, it was necessary to modify their build scripts to use Emscripten to target Wasm. For some libraries, it was sufficient to simply replace calls to the `gcc` compiler with calls to `emcc`, but for other libraries, attempting to compile with `emcc` produced compile time errors. These errors were usually either caused by the usage of a native feature such as a system call not supported by Emscripten, or by the source code defining specific implementations for specific architectures and not recognizing `_OS_EMSCRIPTEN_` as a supported architecture. Patching these libraries was relatively straightforward, except for `LibUV` which required additional modifications.

`LibUV` is a library that enables cross-platform asynchronous IO. Julia strives to support as many platforms as possible, but it can be difficult to maintain different implementations of functionality for each platform that relies on platform-specific APIs. For example, supporting Windows can be particularly tedious when Windows APIs differ from those offered by macOS and GNU/Linux. Usually, platform-specific code is implemented by using pre-processor directives that include or exclude blocks of code based on compiler variables. Julia uses variables such as `_OS_WINDOWS_` and `_OS_LINUX_` to declare platform-specific implementations but using these variables to include or exclude code makes the Julia compiler harder to maintain and greatly

decreases code readability. LibUV solves this problem by providing a single cross-platform API and internally picking the correct implementation based on the platform defined at compile time.

WebAssembly is not a platform currently supported by LibUV, but since the Emscripten compiler attempts to emulate a number of POSIX APIs, it is possible use the LibUV's POSIX implementations for most operations, with the addition of a small `emscripten.c` file for any functions that have Emscripten-specific implementations. For this work, it was not necessary to implement Emscripten support for all LibUV features. The only features that were implemented were those used directly by Julia, specifically the methods `uv_exepath`, `uv_get_available_memory`, `uv_get_free_memory`, `uv_get_total_memory`, and `uv_load_avg`. With these modifications, `LibUV` was successfully compiled as a Wasm library.

Building Julia Internal

Once all the requisite dependencies were compiled as Wasm static libraries, it was possible to begin compiling `libjulia-internal`. Similar to how the libraries were compiled, it was necessary to modify the Julia build scripts to use the Emscripten toolchain in place of the default C compiler. This was done with a custom `Make.user` file, which overrides all the necessary build settings. It sets the compilation target to `wasm32-unknown-emscripten` and sets all the compiler executables to those provided by Emscripten. This works surprisingly well. In fact, with the correct compiler options set, it was possible to compile the latest version of `libjulia-internal` to Wasm with no source code modifications. That is, until the build script called for self-execution.

Build script self-execution is a common technique used when compiling certain complex programs, such as the Julia runtime. Essentially, some subset of the final build is compiled to an executable, and that executable is itself called as part of subsequent build processes. Self-execution can be a challenge when cross compiling to a target architecture different from the build architecture, as the intermediate executable will be built for the target architecture and will not be runnable on the

host machine running the build. In this work, all builds were run on a Linux host machine, with Emscripten-Wasm as the target architecture. In this case, the build script for the Julia runtime compiles a parsing library, `libflisp`, as an executable, and then immediately attempts to run that executable. Since the current build target is Wasm, `libflisp` is compiled as a Wasm module, which cannot be executed on the Linux host.

The solution to build self-execution when cross compiling is to separately build all self-executing components for the Linux host architecture and then use those executables instead of the ones built for the Wasm target architecture. This can be accomplished by creating a separate native Linux build of Julia, and then setting the build variable `FLISP_EXECUTABLE_release` to the path of the flisp executable in that native build.

At this point, `libjulia-internal` could be successfully compiled as a Wasm static library.

Building the System Image

The last required dependency of a functional build of the Julia runtime is the System Image. As detailed in section 4.3.1, the sysimg is just another library dependency, except it is built from Julia source code instead of from C/C++. This means that we can't use Emscripten to compile in same way all the other libraries have been compiled. Instead, this is another case that requires build script self-execution.

In a native build of Julia, the sysimg is built by loading `libjulia-internal` and `libjulia-codegen` into an executable and passing each source file from Julia Base to be compiled by that executable. This strategy will not work for a Wasm cross-compiled build for two reasons. First, as before, the libraries and executables built targeting Wasm cannot be executed on the Linux build system. In addition, `libjulia-codegen` is required to build the sysimg, which is the part of the compiler not built for this runtime-only build.

We use the same strategy as with flisp to build the sysimg: use a native build. However, unlike with flisp where all that was needed was an executable compatible

with the host build architecture, the sysimg must be built as a Wasm library. This cannot be easily done with a native Linux build, as the native build of `libjulia-codegen` can only compile the sysimg targeting the same native architecture. To bypass this constraint, we use a technique introduced in prior work by Keno Fischer [1] to build the sysimg as LLVM bytecode.

When programs are compiled using the LLVM compiler infrastructure, source code is first converted to LLVM IR, then lowered to architecture-specific LLVM bytecode before finally being converted to architecture-specific assembly. The critical detail here is that while assembly code will always be architecture specific and never cross compatible with other assembly languages, it is possible for bytecode intended for different architectures to be compatible if the architectures are sufficiently similar. In his prior work, Fischer [1] showed that the Julia sysimg compiled for 32-bit Linux (i686) was compatible with 32-bit Wasm. This allows us to use a native Julia build to produce a sysimg using the command `make julia-sysimg-bc`. The file produced has the metadata label architecture: `i686-linux-gnu`. By changing this label to architecture: `wasm32-unknown-unknown-wasm`, the sysimg can be linked by Emscripten with no errors.

Building an Executable WebAssembly Module

With all the requisite components built as static Wasm libraries, the final step is to link everything together and call the necessary initialization logic. For normal builds of Julia, the initialization logic is contained in `libjulia`, the Julia loader. The loader is responsible both for dynamically linking all dependencies, including the sysimg, and for initializing the runtime. Since `libjulia` is excluded from this Wasm build, all the initialization logic must be re-implemented, excluding the code for dynamically linking dependencies that have instead been statically linked.

For much of the initialization codepath, this is as simple as copying the implementation of `jl_load_repl` into a separate source file and removing any parts that call `load_library`, as all libraries have already been statically linked. At this point, Emscripten can be used to create an executable Wasm module which can then be

run in a web browser. In this state, the Wasm module can echo constant values, but attempting to parse any expression causes the module to crash.

Initializing the System Image

To parse Julia expressions, it is not sufficient to just statically link the sysimg - it must be properly loaded. When linked statically, the symbols for all functions defined in the sysimg are loaded correctly, but critical global variables are not initialized. The logic to load a sysimg exists in `src/staticdata.c`, but this implementation heavily relies on dynamic linking and does not function correctly when the sysimg is statically linked. Referencing prior work by Fischer, [1] `src/staticdata.c` was modified to support loading a statically linked sysimg.

4.3.4 Julia Codegen in WebAssembly

JIT Compilation in WebAssembly

Building the full Julia compiler as a Wasm module is significantly more challenging that just building the runtime without codegen. As discussed in section 2.2, JIT compilation relies on runtime dynamic linking and indirect function calls, both which are restricted when running in WebAssembly. Prior work [3] has demonstrated that it is possible to implement a JIT compiler in Wasm, but this work has not yet been replicated using Emscripten.

In addition, `libjulia-codegen` requires a full functioning copy LLVM. LLVM is a significantly more complicated dependency when compared to the other dependencies built for `libjulia-internal`, and it was anticipated that significant work would be required to build LLVM as a Wasm module. Fortunately, there exists prior work in the Emception project that addresses this issue.

LLVM in WebAssembly

Emception [2] is an open source project that bundles the entire Emscripten compiler toolchain as a single WebAssembly module, including LLVM compiled to Wasm.

Using a custom build script and the patches to LLVM included in this project allows us to build `libLLVM.a` as a static Wasm library. This enables us to compile and link `libjulia-codegen`, which is a significant step towards a full Julia JIT in Wasm.

Building Julia Internal

`libjulia-codegen` was compiled to WebAssembly using Emscripten with the same strategy used to compile `libjulia-internal`. Surprisingly, it was not necessary to make any modifications to the codegen source code to enable Wasm compatibility.

Dynamic Linking with Emscripten

As mentioned in section 4.3.2, Emscripten recommends that dynamic linking be avoided whenever possible. However, in order to link JIT-compiled outputs to the original Wasm module and make them callable by the runtime, some form of dynamic linking is required. To enable run time dynamic linking in Emscripten, we pass the option `-sMAIN_MODULE` when compiling and linking the Wasm module. This flag enables implementations for the dynamic linking functions `dlopen` and `dlsym`, the API used internally by `libjulia` to load the `sysimg`. Implementing the necessary logic to load the outputs of the JIT compiler running in Wasm is one area of future work for this thesis.

Chapter 5

Results and Conclusion

Overall, this thesis was successful in demonstrating the feasibility of building a Julia JIT compiler that runs entirely in WebAssembly. While it fell short of providing a fully functional prototype runnable in a web browser, multiple critical components have been built, laying the foundation for future work expanding Julia's Wasm support.

Goal	Status
Static compilation of Julia functions to Wasm	*complete*
Execute Julia Wasm functions as Spark UDFs	*complete*
Compile *libjulia-internal* and all dependencies to Wasm	*complete*
Build a Wasm executable of the Julia runtime	*in progress*
Compile *libjulia-codegen* and all dependencies to Wasm	*complete*
Build a Wasm executable of the Julia JIT compiler	*future work*

Successful static compilation of Julia functions using `GPUCompiler.jl` demonstrates that the Julia language is capable of being compiled to WebAssembly using LLVM. The positive results from the Spark UDF experiment shows that small, statically compiled WebAssembly modules can be useful for data analytics applications.

Furthermore, the extensive work done to compile components of the Julia compiler to WebAssembly shows that it is indeed feasible to build this compile as a standalone Wasm module, and lays the groundwork for future experimentation.

Future Work

Further experimentation is required to optimize Spark partition size when executing Wasm UDFs. Smaller partition size increases parallelism but potentially increases overhead by instantiating more Wasm modules.

Additional work is also required to properly initialize the statically linked sysimg, and to integrate the Julia JIT with Emscripten dynamic linking.

Appendix A

Code

The full code for this work is hosted on several GitHub repositories.

- Wasm Static Compilation: https://github.com/rhuffy/GPUCompiler.jl
- Spark Wasm UDFs: https://github.com/rhuffy/spark-wasm-udf
- Julia Runtime in Wasm: https://github.com/rhuffy/julia-wasm

Bibliography

[1] Keno Fischer. Julia-wasm. https://github.com/Keno/julia-wasm, 2020.

[2] Jorge Prendes. Emception. https://github.com/jprendes/emception, 2022.

[3] Andy Wingo. Just in time code generation within webassembly. https://wingolog.org/archives/2022/08/18/just-in-time-code-generation-within-webassembly, 2022.