UNIVERSITY OF HELSINKI

https://helda.helsinki.fi

Bringing WebAssembly Up to Speed with Dynamic Linking

Mäkitalo, Niko; Bankowski, Victor Ludvig Mathias; Daubaris, Paulius; Mikkola, Risto Mikael; Beletski, Oleg ...

2021

http://hdl.handle.net/10138/335259

Mäkitalo, N, Bankowski, V L M, Daubaris, P, Mikkola, R M, Beletski, O & Mikkonen, T 2021, Bringing WebAssembly Up to Speed with Dynamic Linking. in Proceedings of the 36th Annual ACM Symposium on Applied Computing . ACM, pp. 1727-1735, Annual ACM Symposium on Applied Computing, Korea, Democratic People's Republic of, 22/03/2021. https://doi.org/10.1145/3412841.3442045

Downloaded from Helda, University of Helsinki institutional repository. https://helda.helsinki.fi
This is an electronic reprint of the original article.
This reprint may differ from the original in pagination and typographic detail.
Please cite the original version.

Bringing WebAssembly Up to Speed with Dynamic Linking

Niko Mäkitalo
University of Helsinki
Helsinki, Finland
niko.makitalo@helsinki.fi

Victor Bankowski
University of Helsinki
Helsinki, Finland
victor.bankowski@helsinki.fi

Paulius Daubaris
University of Helsinki
Helsinki, Finland
paulius.daubaris@helsinki.fi

Risto Mikkola
University of Helsinki
Helsinki, Finland
risto.mikkola@helsinki.fi
risto.m.mikkola@gmail.com

Oleg Beletski
Huawei Technologies
Helsinki, Finland
oleg.beletski@huawei.com

Tommi Mikkonen
University of Helsinki
Helsinki, Finland
tommi.mikkonen@helsinki.fi

ABSTRACT

WebAssembly is a new technology that aims at portable compilation target for various programming languages. The goal is to support deployment on the web for client and server applications. While the technology itself is independent from the browser, majority of the implementations are browser-based, and hence the associated use cases are limited. In this paper, we study the use of WebAssembly outside the browser. In particular, we are interested in partitioning WebAssembly applications into modules and linking them during execution allowing reductions in memory consumption, binary size, and compilation and startup time.

CCS CONCEPTS

• **Software and its engineering → Modules/packages**; *Software design tradeoffs*; *Software libraries and repositories*;

KEYWORDS

dynamic linking, dynamic loading, execution time, runtime environment, modularization, WebAssembly, Wasm.

ACM Reference Format:
Niko Mäkitalo, Victor Bankowski, Paulius Daubaris, Risto Mikkola, Oleg Beletski, and Tommi Mikkonen. 2021. Bringing WebAssembly Up to Speed with Dynamic Linking. In *Proceedings of ACM SAC Conference (SAC'21).* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

The demand for cross-device programming languages has been constantly increasing. The era of mobile and pervasive computing has been followed by the the era of ubiquitous computing, meaning that more and more heterogeneous devices get integrated in our surroundings, as envisioned in [20]. Furthermore, these devices as well as facilities that support them online require very different programming approaches [10, 18].

During the years, various approaches have been taken for having a common programming language and a runtime environment that allows using the language in different contexts. The original *write once, run everywhere* promise of Java [5] has recently been followed by JavaScript [4] (ECMAScript [3]) and its numerous runtime environments. Due to their popularity, these solutions have improved the portability of code, allowing designs like Node.js[1] that allow using JavaScript outside the context of the Web [12]. However, JavaScript was originally designed for web browser environments to event-based programming, and thus some of its design decisions are not well-suited for other environments [11].

A new platform independent format called WebAssembly (Wasm) has been recently introduced [6, 13]. It is a portable binary instruction format for stack-based virtual machines[2] with key features that enable running untrusted programs securely[3] while still maintaining fast execution[4,5]. Wasm already has broad support among major browsers (Firefox, Chrome, Safari, Edge). The format is designed with network transfer in mind and is fit for streaming compilation scenarios. Wasm is designed to be a compilation target, which can support several languages: C, C++ and Rust can already be compiled to Wasm. In the last few years, running Wasm code outside the browser has started to gain interest among researchers and developers [21][6,7], which was also manifested in [1]. On the surface, Wasm is a perfect fit for a cross-platform applications – the virtual machine required to run applications has a small footprint, and their performance is near to native.

In this paper we study how Wasm modules can be dynamically linked outside web browser's context to create modular applications. With this capability, platform specific features could be isolated into specific modules while generic functionality is implemented in modules that would be shared across platforms. This decomposing of monolithic applications into more modular ones allows faster starting time, if applications can be loaded in pieces. The main benefit of our approach is that it enables building modular and dynamic

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SAC'21, March 22-March 26, 2021, Gwangju, South Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/xx.xxx/xxx_x

[1] https://nodejs.org/ accessed Sept. 8, 2020.
[2] https://webassembly.org/docs/portability accessed Sept. 15, 2020
[3] https://webassembly.org/docs/security accessed Sept. 15, 2020
[4] https://github.com/wasm3/wasm3/blob/fcad718a29866e5b285a8db288c58f1a72a4616d/docs/Performance.md accessed Sept. 15, 2020.
[5] https://medium.com/@torch2424/webassembly-is-fast-a-real-world-benchmark-of-webassembly-vs-es6-d85a23f8e193 accessed Sept. 15, 2020.
[6] https://webassembly.org/docs/non-web accessed Sept. 15, 2020.
[7] https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface accessed Sept. 15, 2020.

applications that can be extended during execution. Moreover, as a long-term vision, because the different platforms are running essentially the same Wasm code, it would become possible to migrate the applications between different computers, thus supporting the liquid software paradigm [7, 19].

We identify execution time dynamic linking as a missing key feature of Wasm. Without such facility, a problem of emerges that applications need to be loaded in full before the first line can be run. This results in delayed startup time and consequent downgraded user experience. As a practical solution, we present a system for loading and linking Wasm modules on the fly, which allows hiding some of the loading times from the end user. Unlike most work in the Wasm context, we seek to to build applications outside web browser. To this end, we demonstrate and evaluate the approach outside the web browser environments, and describe how the applications can be run in resource-constrained devices.

The rest of this paper is structured as follows. In Section 2, we give background regarding Wasm and its requirements for execution time dynamic linking. Furthermore, we introduce our envisioned system and motivate why the current related, browser-based solutions cannot be leveraged outside the web browser. In Section 3, we present our approach and describe its most important implementation details. In Section 4, we evaluate the solution. In Section 5, we discuss about how well our solution fits to its purpose of running modular Wasm applications outside web browsers. Moreover, we mention the future work to be expected. Finally, in Section 6, we summarize this paper.

2 MOTIVATION AND BACKGROUND

WebAssembly was initially designed to be compatible with the Web [22]. Therefore, Wasm applications can call into and out of the JavaScript context and access browser functionality through the same Web APIs accessible from JavaScript. Hence it is possible to use Wasm for increased performance [6] as well as for other, more indirect needs, such as code protection via obfuscation [17]. When using Wasm inside the browser, there is a sophisticated, well-defined API for accessing Wasm functions from JavaScript. At the core of this API is the JavaScript object *WebAssembly*, which can be used to load and instantiate modules, to allocate memory, and to handle error situations that may emerge when loading a new module or when accessing its functions.

For web pages and browser applications, which have already become overly complex as is [2], embedding Wasm to JavaScript is an option that does not add much memory related or performance constraints. However, for applications that run in devices with restricted memory, such embedding uses too much memory. Despite the increasing computing capacity of chips, it is still expected that the future networks include memory and performance related challenges [15] as many devices have limited memory [14]. Moreover, in the context of IoT systems, computers in general have diverging performance capabilities, ranging from almost bare metal in sensors to cloud systems where everything is virtualized, sometimes in several layers [18].

In addition, rapid response times are essential for both user experience and business reasons. In general, frequent users prefer response times of less than a second for most tasks, and as response time decreases, productivity increases [16]. In contrast, for instance Amazon has found out that every 100ms increase in the page load time would decrease sales by 1%, while for Google a 500ms increase in delay of search results display time would reduce revenue by 20% [8].

2.1 WebAssembly and Dynamic Linking

Wasm programs are organized into modules, which are the unit of deployment, loading, and compilation [6]. Each module can contain definitions for types, functions, tables, memory areas, and global variables. These definitions may be imported or exported. When Wasm modules are instantiated, the host program must provide all imports required by the module. Essentially, this means that complete linking takes place as the host program instantiates all the modules. This results in an application with size and startup time similar to a monolithic, self-contained application. In many scenarios, Wasm binaries need to be downloaded over a network or compiled into native code on the target machine, which will result in increased start-up time. For example, downloading a 16MB application on a 1MB/s network connection will take roughly 16 seconds, which is a significant waiting time. However, Wasm also offers facilities for establishing more advanced concepts, such as streaming compilation and execution time dynamic linking that can be used to parallelize and distribute the start-up process to minimize the delay at start-up.

A Wasm module can import and export definitions from and to the host environment. The definitions include functions, tables, memory areas. Functions can be invoked to perform actions, and function references can be stored in tables, thus emulating function pointers. Function references in a table can be changed during execution. They can have different signatures, which are validated on each function call to ensure integrity. Depending on the needs, function tables can be modified or changed in size even from outside of a module instance. Wasm memory is of linear layout, and it can be shared between multiple module instances [9]. Similarly to tables, the size of an allocated memory area size can be modified. Tables and memory areas can be shared by multiple module instances, and the host can provide definitions of its own as well as module exports as imports to module instances.

To enable execution time dynamic linking, one must be able to share a table through which function pointers can be shared across multiple module instances. Similarly to sharing a table, a memory is also a necessary part of a scheme which enables memory access without any overhead. When two modules are linked, a module instance requiring another module provides its memory to the required module through an export. The same happens to the module instance's table. References to the required module's functions can then be stored into the table allowing them to be called by any module's instance having access to it.

2.2 Envisioned system

We envision a system, illustrated in Figure 1, which would allow an initial base module instance to retrieve additional modules from online repositories on-demand and dynamically link them during execution. The base module instance could even use a standard

interface to detect the characteristics provided by its environment, and dynamically load functions to adapt to this environment.

Currently, Wasm outside of the browser does not offer any execution time dynamic linking capabilities [8]. Therefore, all parts of an application must be present to run it, which can introduce long loading and startup times, directly related by the binary size of a module.

To avoid this restriction, we propose a execution time dynamic linking solution, where the execution of an application can start even if all the modules are not present. This would result in faster startup times, whereas additional features could be loaded on an on-demand basis. In addition to faster startup time, the approach also has benefits related to memory usage. As the monolithic application would be split into separate modules, an application could only load the necessary functionality at the time when it is needed saving the memory. The memory could also be monitored and unused module instances could be released whenever there is a need to load additional functionality. This is particularly important for embedded devices where resources are constrained.

3 DYNAMIC LINKING OUTSIDE THE BROWSER

Following the previously described concepts, we decided to use execution time shared-everything linking approach depicted in Figure 2. The approach enables us to share functionality between module instances by populating a global table of function references. The table is accessible to any module instance that is linked in the executable, and it allows cross-module instance function calls through the references. The module instances also share a memory area, which means that they can access each other's allocated memory directly.

Inspired by Linux, in our design, execution time dynamic linking is handled with three functions, *dlopen*, *dlsym*, and *dlerror*. The *dlopen* function is used to load a shared library, *dlsym* returns the index where a symbol is placed in the table, and *dlerror* allows fetching error messages. In addition, the implementation defines three global objects. *INSTANCES* stores a vector of all module instances. *LINKER* is a manager for imports and exports of module instances that must be available throughout the lifetime of a program as it needs to be accessed by *dlopen* and *dlsym* functions at various points of the program's lifetime. The third object, *DLERROR*, stores the information of an error buffer where error messages from *dlopen* or *dlsym* are written. Both dlopen and dlsym, return zero values in case of an error.

3.1 Implementation

The implementation consists of a host program, which uses the Wasmtime[9] runtime to run the Wasm binaries and interacts with the runtime to provide dynamic linking capabilities. Wasmtime provides a high-level API and useful features, such as utilities for managing module instance imports and exports. For our use a key structure is *Store*[10], where all created instances and items, such as functions, globals and tables, are attached. The items of *Store* are freed only once there are no references to the store or to any of its items. Another important structure is *Linker*[11], a name-based resolver where names are dynamically defined and later used to instantiate a module. A method of a *Linker* used to instantiate a module will automatically select the required imports, or, if this cannot be completed, return an error.

As a part of the process of dynamically loading a module, the details of the module's memory and table size requirements must be acquired. In our implementation we use Emscripten[12], an open source compiler toolchain for compiling C, C++, or other LLVM based code to WebAssembly, to compile our modules [23]. The rationale is simple – currently only Emscripten compiled modules can produce dynamically linkable modules[13,14]. The generated modules include a "dylink" section which contains the module's required memory size, memory alignment, table size, and table alignment. Figure 1 shows the presence of the *dylink* section in a shared module compiled with *Emscripten*.

The host program begins by creating new *Store* and *Linker* instances. The *Linker* is then populated with references to the host defined functions (*dlopen*, *dlsym*, and *dlerror*). The main module's Wasm binary, which is the starting point of an application, is then loaded into memory and compiled into a module. The module's imports are inspected, and, if required, a table and memory is created by the host, and references to them are stored into the *Linker* instance. The *Linker* instance is then used to instantiate the module automatically resolving required imports. After the module is fully instantiated, its exports are stored to the *Linker* data structure under a module's name. Error handling is then initialized by creating a buffer to store error messages with the *alloc* function. The position and size of the buffer and a reference to the instance's memory are stored for later use. The *Linker* is also stored to a global variable for later access. Finally, the main function calls the main instance's entry function, thus starting the execution.

3.1.1 dlopen. To load a module, the *dlopen* function is called. The function accepts one parameter, the name of the desired library to be opened. The function's call sequence is depicted in Figure 3. When called from an instance, the function retrieves memory from a global instance of a linker and retrieves the name of the library supplied as an argument to *dlopen* by accessing the memory. After acquiring the name of the desired module, *dlopen* searches if an instance already exists in the global instance vector defined in the host. If an instance is found, the index of the instance in the vector is returned as it is the identifier of an instance. If an instance was not found, it is assumed that a file corresponding to that particular Wasm module's instance is placed in the location relative to the executable. The file is searched for by concatenating the module's name with the Wasm file extension. If the file of a module is found, the file is opened, read and compiled into a module, which then has to be linked with the other module instances. Otherwise, the linking process raises an error.

[8] https://github.com/bytecodealliance/wasmtime/issues/1934

[9] https://github.com/bytecodealliance/wasmtime accessed Sept. 8, 2020.

[10] https://docs.rs/wasmtime/0.19.0/wasmtime/struct.Store.html accessed Sept. 8, 2020

[11] https://docs.rs/wasmtime/0.19.0/wasmtime/struct.Linker.html accessed Sept. 8, 2020

[12] https://emscripten.org/

[13] https://github.com/emscripten-core/emscripten/issues/11954 accessed Sept. 8, 2020

[14] https://github.com/rust-lang/rust/issues/60231 accessed Sept. 8, 2020.

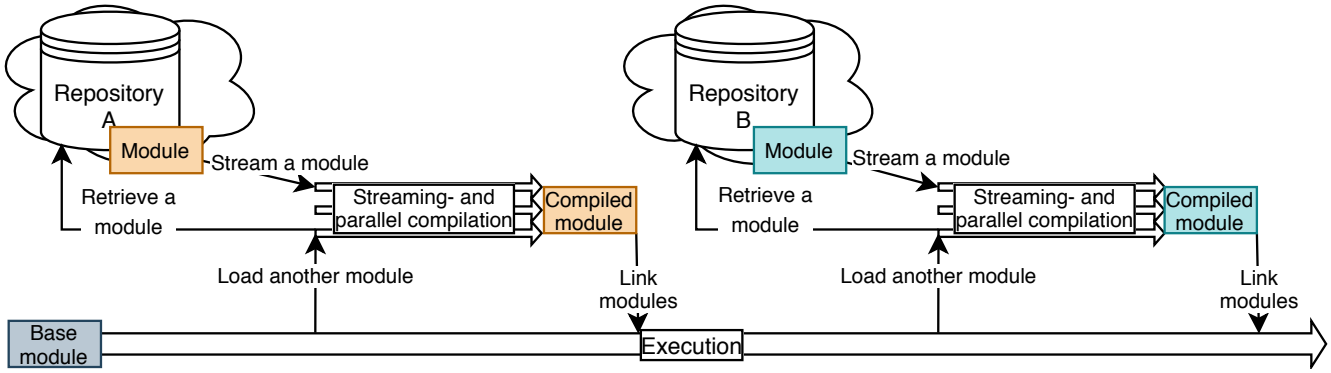

Figure 1: Execution of a modular program where modules are loaded and linked during the execution.

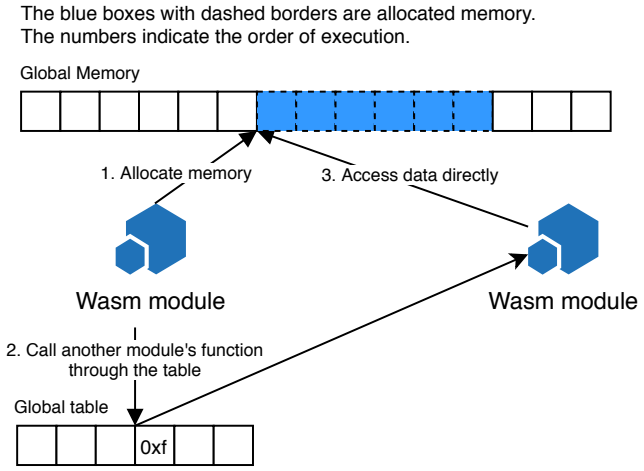

Figure 2: Memory and table usage in cross-module instance call of shared-everything dynamic linking approach.

Listing 1: Disassembly of a module with the *dylink* section.

```
1 Custom start=0x0000000a end=0x00000016 (size=0x0000000c) "dylink"
2 Type start=0x00000018 end=0x00000029 (size=0x00000011) count: 4
3 Import start=0x0000002c end=0x000000b1 (size=0x00000085) count: 7
4 Function start=0x000000b3 end=0x000000b8 (size=0x00000005) count: 4
5 Global start=0x000000ba end=0x000000c0 (size=0x00000006) count: 1
6 Export start=0x000000c2 end=0x0000010f (size=0x0000004d) count: 4
7 Code start=0x00000112 end=0x0000032f (size=0x0000021d) count: 4
```

After compiling a module in *dlopen*, it is linked. To acquire the required linking information from the *dylink* section, the module binary is parsed. Then, a chunk of memory is allocated using the *alloc* function by providing the required memory size from the linking information. Similarly, the table is grown by using table size from the linking information. After these steps are completed, the module is instantiated. After the instantiation of a module, any initialization steps required by the module are executed. The instance is then pushed to the global vector of instances and the numeric identifier is returned from *dlopen*, called a *handle*.

3.1.2 dlsym. Figure 4 illustrates the process of resolving a function using *dlsym*. The method takes two parameters, a handle to an instance of which symbol must be resolved and the name of a symbol, which in our case is always a function's name. Firstly, a function name is retrieved by accessing the instance's memory as seen in the figure. Secondly, an instance is retrieved from the global vector of instances using the handle supplied to *dlsym* which is an index to the vector. The instance is accompanied by a map of symbol names and their table indices. The map is accessed with the symbol's index. If a function is found, its index is returned as a result of *dlsym*. Otherwise, the required function is retrieved from the instance and added to the shared table by growing it by one unit initialized to a reference to the function. The index of the function reference is inserted into the map of that particular instance and returned.

3.1.3 dlerror. The *dlerror* function is used to check if an error occurred and to access error messages. The main function of the host initializes a buffer for the error messages. Error handling information structure (Figure 2) is stored by the host, and it contains the error message buffer's size, its location in memory, a flag indicating if an error has occurred and a reference to the memory object shared by the instances. When an error occurs in any of the three host functions, *write_error* function is called to write a null terminated error message to the error buffer and the error occurrence flag is set to true in the data structure that the host stores. To access the error message, the *dlerror* function is called. The function returns zero if no error has occurred. Otherwise, the position of the error buffer is returned from the structure and the occurrence flag is set to false. The *dlerror* function is typically used to print an error message after *dlopen* and *dlsym* calls return a zero value indicating the occurrence of an error.

3.2 Example Application

To demonstrate the above features, we next present an example application, a chatbot that can dynamically load modules reflecting different personalities (Figure 5). The main module of the application can run on its own as a standalone application. The main module defines and exports shared library functions for others to use. Other modules require the main module and are called side

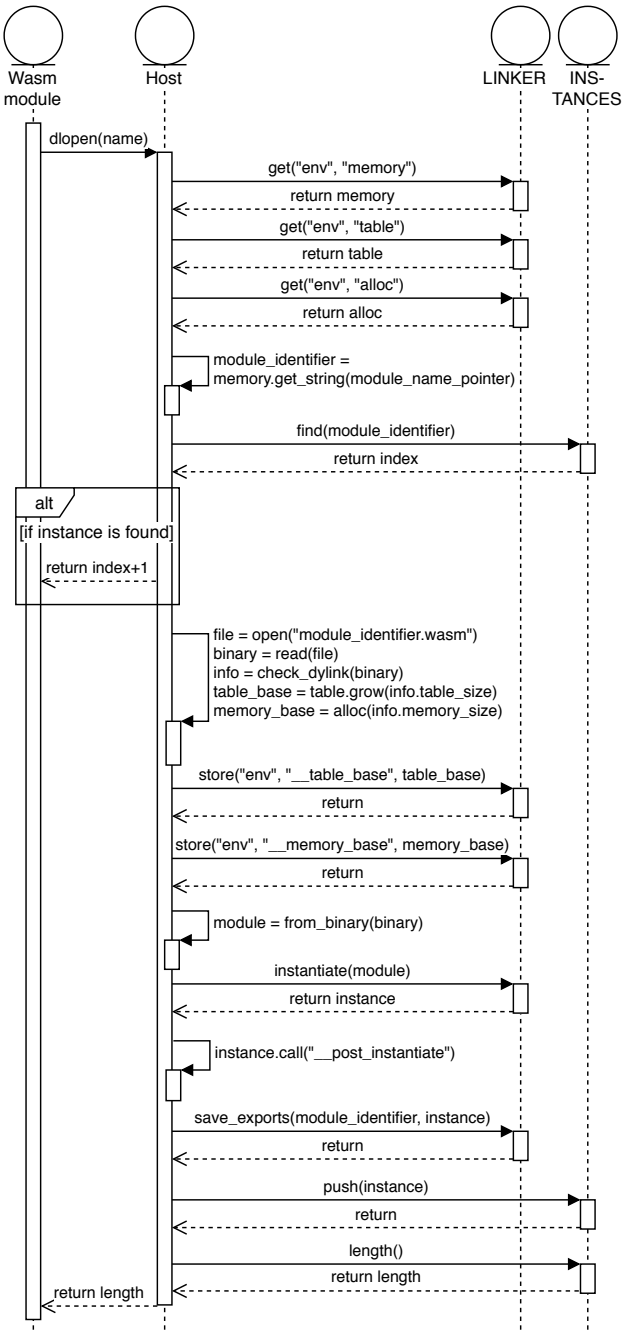

Figure 3: Sequence diagram of a *dlopen* function call.

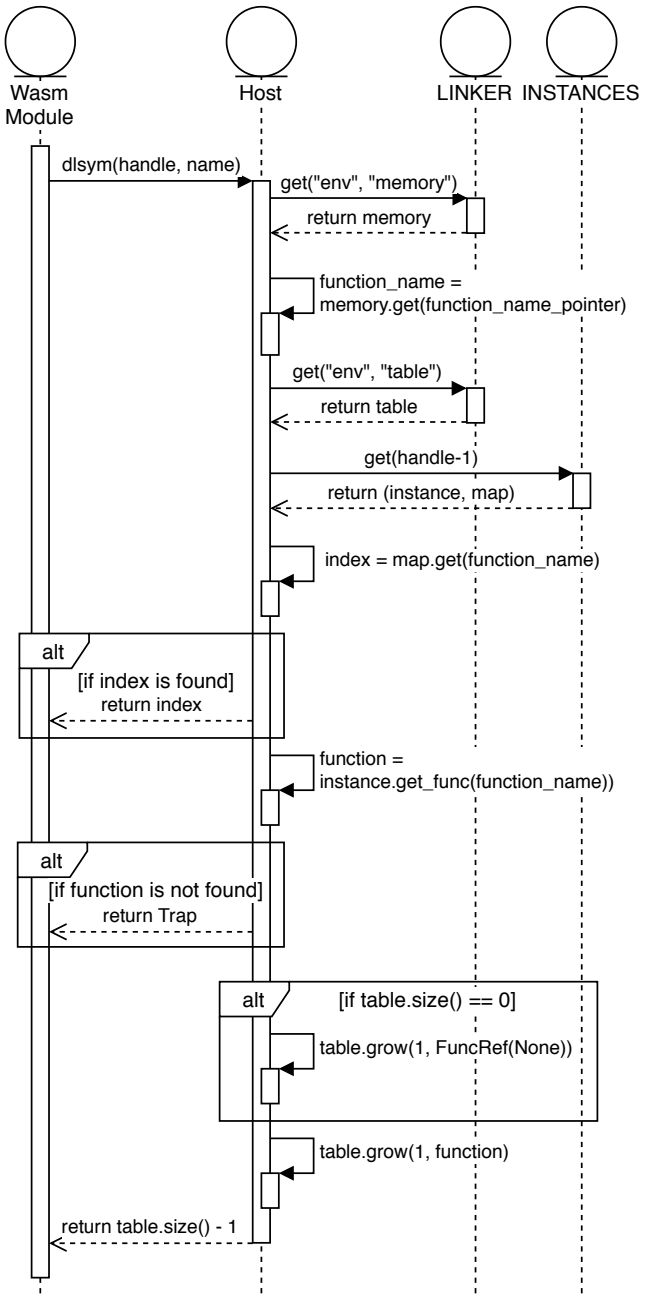

Figure 4: Sequence diagram of a *dlsym* function call.

modules. The side module instances import various functions from the main module instance, such as a memory allocation function. The main instance can load the personalities during execution. Once a personality module is loaded, it can load more modules when user wants to do different actions. The modules present various image processing functions and procedures displaying ASCII art on the screen. All modules linked together share the same resources. Therefore, they are able to pass complex and large data structures between each other with ease and without copying them.

A sample execution of the application is presented in Figure 6. This particular scenario explores the internals of Steve's personality module. In the diagram the user has already started the program and selected to load Steve personality. The program waits for the user's input and receives a command to open a file. The image is loaded into memory and new command is asked from the user. The user then provides a known processing command. As a result, the

Listing 2: Data structure representing error information.

```
struct ErrorInformation {
        size: usize,
        position: usize,
        occurred: bool,
        memory: Option<Memory>,
}
```

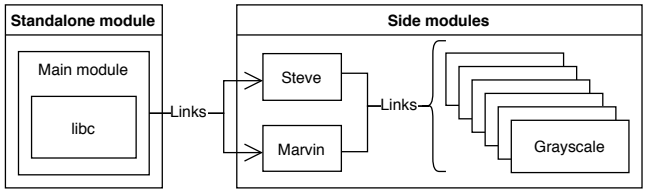

Figure 5: Relationships between the modules.

personality loads the module responsible for that specific processing task using *dlopen*. The processing function of that module's instance is resolved with *dlsym* yielding a pointer to the function. The function is then called with a pointer to the image data, which applies the operation to the previously loaded image. After the side module's function has finished executing, the personality will ask the user for input. The user can apply more operations on the same image, but in this case he decides to save the image. The resulting image is written to the destination file given by the user. The user is then asked for an input and he decides to exit back to the main instance.

A video about the example application is available at[15]. The demonstrator on the video explains the core features of our implementation discussed above, demonstrating also how the dynamic loading of the modules affects the memory consumption, and illustrates how a module is shared between multiple instances.

4 EVALUATION

To evaluate our execution time dynamic linking implementation, we explore the following key metrics: startup time, time of the entire execution of a use case, memory overhead, and overhead caused by the use of our implementation. We analyze the performance by comparing two different application approaches for each of the scenario. The approaches involve measuring the previously described metrics of one monolithic application and a modular application which loads needed functionality using our system.

Listing 3: Wasm compilation flags.

```
-Wl,
--import-table,
--no-entry,
--allow-undefined

-s ALLOW_MEMORY_GROWTH=1
-s ERROR_ON_UNDEFINED_SYMBOLS=0
```

[15]https://youtu.be/gZj3M31ZfuI

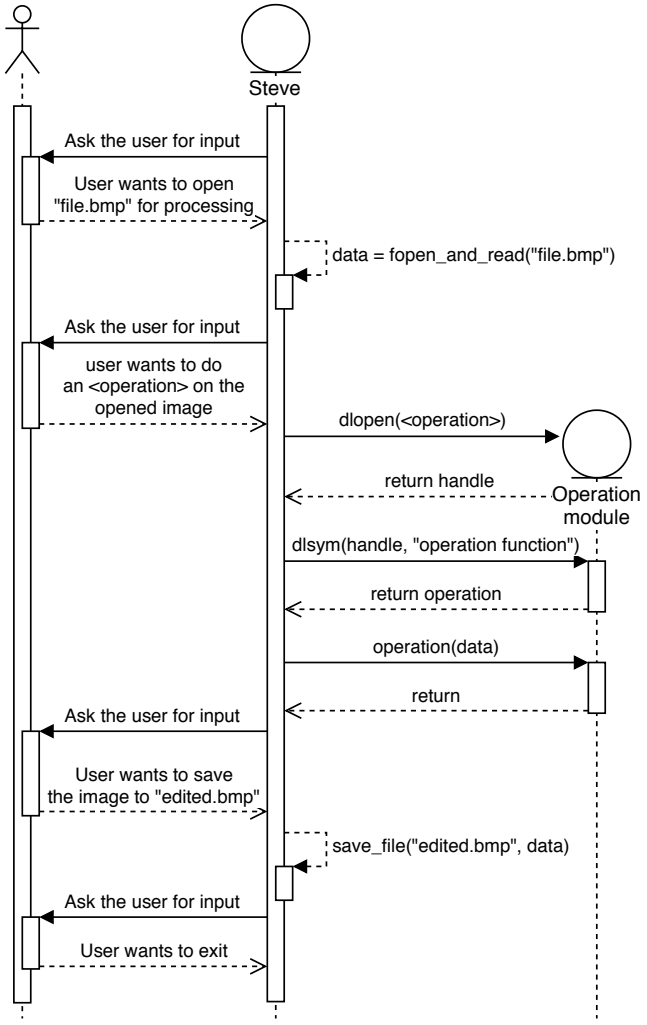

Figure 6: Scenario where a personality module is first loaded, and then it loads an additional module dynamically.

All measurements have been done on 64 bit Ubuntu 18.04.5 LTS with 16 GB RAM and an Intel i5-7200U@2.50GHz x 4 processor. The modules for the measurements were compiled with Emscripten 2.0.0 using the flags in Listing 3. We have decided to omit optimization flags to make the programs comparable. The time and RAM measurements were done with the *time* command available in Linux. We report the maximum resident set size as RAM in all tables. The Wasm runtime used by the implementation for the measurements is Wasmtime 0.19.0.

4.1 Startup Time

The first measurement studies the startup time of a monolithic and modular applications. To assess the startup time of each application we measured the time it takes for the application to start and immediately exit the program without doing any work. Starting the application includes numerous operations, such as reading the entire file required to start the application, compiling it into a module,

Table 1: Startup time measurements.

Approach	Binary Size	Startup Time	RAM
Monolithic	15.2 MB	4.472s	321284 KB
Modular	28.6 KB	0.104s	9852 KB

and, finally, instantiating the module using the runtime's embedding API. In addition, we measure the RAM that the application uses during the startup and the size of the binaries used for the measurements. In this measurement the modular approach does not load any additional modules, as it immediately exits once it has loaded the main application module.

Table 1 summarizes the results of this measurement. Unsurprisingly, there is a considerable difference in the size of the binaries. As mentioned before, using our execution time dynamic linking implementation, an application can be split into separate parts. Therefore, a file required to start a modular application can be significantly smaller comparing it to a monolithic application. As a result, operations such as reading a file, compiling and instantiating a module take significantly less time. As the startup time is directly related to the binary size, this also improves a lot. Finally, the memory that is needed to initialize the modularly loaded application is again only a fraction of its monolithic counterpart.

4.2 Entire execution

Next, we measured the entire execution of a use case where the application starts, executes a single function and then exits. The application consists of thousands of independent functions, of which only one is invoked by the use case.

For these measurements, we had a monolithic configuration and three modular configurations: (1) load a single module and resolve a single function (LS RS), (2) load all modules and resolve all of its functions (LA RA), and (3) load all modules but resolve only single function (LA RS). The monolithic configuration consists of a single module that contains all the application logic. The configuration loads the module and executes a task using only a fraction of the application's logic. The modular application consists of ten 1.5 MB modules that contain the functions split evenly and an initial module's instance that loads and resolves the functionality of the other modules depending on the used configuration. The first modular configuration (LS RS) loads only the necessary module to do its job. This configuration simulates a situation where a specific feature is used immediately after the application is started. The other two modular configurations (LA RA and LA RS) mimic the monolithic approach by loading all the modules available. In addition, LA RA configuration resolves all of the symbols from the loaded module instances while LA RS resolves only the single required symbol. These setups reveal how much overhead the linking system adds to the entire execution compared to the monolithic application approach.

Table 2 presents the total binary sizes of each configuration, the time it took to execute the selected routine, and its memory usage. An outlier in this experiment is LS RS, while other are roughly the same in terms of all the measurement units. The binary size reflects the sum of the sizes of all modules used by that particular configuration. Modular configurations require more memory than the monolithic application, because different amounts of logic are required for loading and resolving the symbols of the additional modules.

Table 2: Full execution time measurements.

Approach	Total Binary Size	Execution Time	RAM
Monolithic	16.000 MB	4.50s	323692 KB
LS RS	2.439 MB	0.48s	47500 KB
LA RA	16.033 MB	4.86s	333140 KB
LA RS	16.030 MB	4.60s	333000 KB

4.3 System overhead

Loading modules and resolving their functions during execution has an overhead compared to a monolithic application. To assess the overhead imposed by the execution time dynamic linking implementation we measure the average time of different function calls. We measure the time it takes to call a function of a linked module and the time it takes to call the functions (*dlopen* and *dlsym*) our implementation provides for Wasm module instances. Two different techniques to call a function of a linked module are measured. One is a direct call by using the import functionality provided by Wasm, and the other is an indirect call through a Wasm table with the use of *dlsym*.

For the overhead measurements we created modules each of which contain a function that accepts an integer as an argument and returns it immediately. Table 3 depicts the measurements consisting of five functions that are called. The first function – *dlopen* – in this particular setup, loads a module of 263 bytes size containing the previously mentioned function. The second function – *dlsym* – resolves the function from a module instance. The third function, specified in table as *symbol*, is resolved from another instance with *dlsym*. The fourth function, specified in the table as *import*, is imported from another instance. The fifth function, specified in the table as *function*, is defined and called within the same module instance and is used as a baseline for a function call. In the measurement, the function under study is constantly called in a loop. The execution time of the loop is divided by the number of calls made to get the average time.

Unsurprisingly, as *dlopen* is clearly the most complex function due to compilation and initialization steps, its execution is orders of magnitude more expensive than the execution of others. In addition, since *dlsym* has to manage data structures related to exports and imports, its execution takes several microseconds. For others, the execution time seems indifferent, although the overhead of calling a function dynamically linked during execution is about 3 times of that of a normal function call.

5 DISCUSSION

Wasm advocates platform independent compilation target that enables creation of modular applications. Here, we use this modularity to improve startup time, but similar mechanisms can be used to isolate platform and application specific features into separate modules. By dynamically loading only the required parts, it is possible to run only the needed parts of complex applications under constrained

Table 3: Function execution time measurements.

Call	Average Time
dlopen	948.6400 μs
dlsym	34.2970 μs
symbol	0.0098 μs
import	0.0048 μs
function	0.0034 μs

devices. Furthermore, modules containing additional features can be loaded on-demand basis.

5.1 Analysis of Results

Despite the current state of execution time dynamic linking in Wasm, we were able to create a system allowing it outside of the browser. We achieved this by using tooling that allows creation of dynamically linkable modules for the browser's environment and by implementing the missing execution time dynamic linking functionality for outside of the browser's context.

Our performance measurements investigated the startup time, memory consumption and binary sizes of different configurations. The results clearly indicate that modular approach reduces the startup time required to begin executing an application significantly, by roughly 98%. Binary size of a starting application has also decreased, as modularity allows one to only load the necessary parts of an application, instead of a monolith containing functionality that is yet to be used.

In addition to the startup time measurements, we investigated how the execution time dynamic linking solution impact the overall execution of a certain routine. The main observation of this particular test is that whenever loading a tenth of a module's functionality to execute a particular task, the execution time is reduced by roughly 90%.

Our implementation introduces a slight overhead. We evaluate the performance of functions provided by the implementation which permit on-demand loading and linking. Moreover, we examine cross-module instance function calls (*symbol*, *import*). We compare the results with the baseline *function* shown in the Table 3. The results reveal that the *dlopen* function takes the longest to execute. This was expected, as the function needs to compile the Wasm binary into a module to be used by the runtime. The *dlsym* function is in general a lot faster compared with the *dlopen* function, but it is not as fast as the cross-module instance function calls and still introduces a slight overhead. Cross-module instance calls are only slightly slower than direct function calls.

5.2 Limitations

An important part of this research was to identify the limitations affecting our produced result. The limitations are divided into four categories: runtime, compilers, Wasm and designed solution.

5.2.1 Runtime. Our work is largely affected by the underlying Wasm runtime. Wasmtime – as well as other Wasm runtimes, too – is progressing quickly. There is no doubt that eventually the runtime capabilities will expand enabling improvement of parts of our solution. As an example, Wasmtime is limited to a number of platforms it can be run on but improvements are made constantly, adding support for more instruction sets on different architectures. As new platforms are supported by the runtime, our solution can then also be ported to these platforms. It is also possible that some parts of our solution could be integrated into the runtime instead of being external features built on top of the runtime. Moreover, the complete unloading is ultimately impossible at the current state of the Wasmtime runtime that we are using. The runtime uses a scheme where all objects are tied to a *Store* object and all objects exist until the store or any objects it references are no longer used. This poses a limitation to the dynamic linking scheme because the *Store* object does not offer any method to remove a reference to a specific instance.

5.2.2 Compilers. Compilers cannot currently generate dynamically linkable modules for outside of the web context or support for it is experimental[16,17]. Additionally, the streaming compilation of Wasm modules appears to be a work in progress for the Wasm runtime we have experimented with[18]. Elimination of these deficiencies will take time and require work on the compiler tooling as well as the runtime, but they are not unsolvable.

5.2.3 Wasm. While the current Wasm specification allows the creation of execution time dynamic linking, there is a proposal for module linking called "Module Types"[19] that can simplify and improve the implementation even further. This proposal notes that dynamic linking capabilities are currently dependent on the host and there is no portable way to dynamically link modules. The proposal suggests adding load-time dynamic linking to the Wasm specification. While this would not implement execution time dynamic linking it could require runtimes to implement features that execution time dynamic linking implementation could use, such as freeing the memory of unused module instances.

5.2.4 Designed solution. Our design has a number of weakness points that should be acknowledged and eliminated. To begin, our prototype implementation provides no way of unloading a loaded module's instance. This is problematic, because no memory is released until the end of the program's lifetime.

Another issue is the overhead of *dlopen*. In our implementation the execution of the program halts until the loaded module is read from disk, compiled and instantiated. This may take a considerable amount of time, as also seen in Table 2. This could be alleviated by using asynchronous retrieval and parallel compilation of the module. Additionally, code analysis could be used to determine when modules would be needed in advance so that little to no overhead is experienced when *dlopen* is actually called. However, this is a major research task in itself that falls beyond the scope of this paper.

Currently, the module instances share a single memory space, which means that they can access each other's allocated memory

[16] https://github.com/emscripten-core/emscripten/issues/11954 accessed Sept. 24, 2020.

[17] https://github.com/WebAssembly/tool-conventions/blob/master/DynamicLinking.md accessed Sept. 24, 2020.

[18] https://github.com/bytecodealliance/wasmtime/issues/1987 accessed Sept. 24, 2020.

[19] https://github.com/WebAssembly/module-linking/blob/master/proposals/module-linking/Explainer.md accessed Sept. 24, 2020.

directly. Because everything is shared there is no concept of private data and it is not possible to hide sensitive data or functionality from different instances, which can threat privacy and security [9].

5.3 Future Directions

Figure 1 depicts our envisioned system, where Wasm modules are distributed to different platforms. The modular nature of Wasm allows creation of small modules, from which developers can then compose applications, similarly to the Node.js package ecosystem. The research has so far focused on designing a mechanism for dynamically loading modules on-the-need basis; ecosystem perspective as well as considerations regarding the standardization of the module structure or features has so far been overlooked.

Next, we want to reduce the memory footprint by storing the modules in multiple external repositories and implementing the capability to manage them at execution time. This requirement is also directly related with the action of module instance unloading. Furthermore, we plan to study the ways the system can be optimized. These include (1) loading modules before they are required so that the overhead would be as minimal as possible, (2) detecting and unloading unused instances so that resources of a system could be managed, and (3) caching and sharing modules between multiple applications. We also hope that streaming and parallel compilation capabilities will be implemented and improved by the available Wasm runtimes as our use case could greatly benefit from these features.

6 CONCLUSIONS

In this paper, we have explored the use of WebAssembly for building cross platform modular applications outside the browser. We believe that the combination of Wasm and execution time dynamic linking can revolutionize the way we build cross platform software. These applications could potentially run anywhere from browsers, desktop computers and phones to embedded devices. The ability to compose applications from modules allows the creation of an ecosystem where components can be developed independently, shared and combined to create various applications. A particular key challenge we have tackled is the implementation of dynamic linking in non-browser environment for Wasm. In the technical sense, this solution can improve startup time of Wasm applications; on a broader scope, such an approach can foster a developer ecosystem.

As a technical contribution, we presented a system that builds upon the previously introduced Wasm concepts that are ultimately inspired by the Linux dynamic linking approach. We provide functionality to load Wasm modules of an application and link the desired functionality during execution. We built a proof of concept implementation of execution time dynamic linking for Wasm in outside of the browser environments, which to our knowledge is one of the first implementations. We performed measurements using the implementation and they show that it achieves expected improvements for program footprint with low overhead on function calls. The work lays a foundation for future work to take full advantage of modular Wasm applications that could adapt to their environment through dynamic loading.

REFERENCES

[1] David Bryant. 2020. WebAssembly Outside the Browser: A New Foundation for Pervasive Computing. In *Keynote at ICWE'20, June 9-12, 2020, Helsinki, Finland.*

[2] Michael Butkiewicz, Harsha V Madhyastha, and Vyas Sekar. 2013. Characterizing web page complexity and its impact. *IEEE/ACM Transactions on Networking* 22, 3 (2013), 943–956.

[3] ECMA. 2020. 262: Ecmascript language specification. *European Association for Standardizing Information and Communication Systems, https://www.ecma-international.org/publications/standards/Ecma-262.htm* (2020).

[4] David Flanagan and Will Sell Like. 2006. JavaScript: The Definitive Guide, 5th.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java language specification.* Addison-Wesley Professional.

[6] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 185–200.

[7] John Hartman, Udi Manber, Larry Peterson, and Todd Proebsting. 1996. *Liquid software: A new paradigm for networked systems.* Technical Report. Technical Report 96.

[8] Ron Kohavi and Roger Longbotham. 2007. Online experiments: Lessons learned. *Computer* 40, 9 (2007), 103–105.

[9] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th {USENIX} Security Symposium ({USENIX} Security 20).* 217–234.

[10] Niko Mäkitalo, Francesco Nocera, Marina Mongiello, and Stefano Bistarelli. 2018. Architecting the Web of Things for the fog computing era. *IET Software* 12, 5 (2018), 381–389.

[11] Tommi Mikkonen and Antero Taivalsaari. 2007. *Using JavaScript as a real programming language.* Sun Microsystems, Inc.

[12] Patrick Mulder and Kelsey Breseman. 2016. *Node. js for embedded systems: using web technologies to build connected devices.* " O'Reilly Media, Inc.".

[13] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 23–42.

[14] James Noble and Charles Weir. 2001. *Small memory software: patterns for systems with limited memory.* Addison-Wesley Longman Publishing Co., Inc.

[15] Ella Peltonen, Mehdi Bennis, Michele Capobianco, Merouane Debbah, Aaron Ding, Felipe Gil-Castiñeira, Marko Jurmu, Teemu Karvonen, Markus Kelanti, Adrian Kliks, et al. 2020. 6G White Paper on Edge Intelligence. *arXiv preprint arXiv:2004.14850* (2020).

[16] Ben Shneiderman. 1984. Response time and display rate in human performance with computers. *ACM Computing Surveys (CSUR)* 16, 3 (1984), 265–285.

[17] Jian Sun, DingYuan Cao, XiMing Liu, ZiYi Zhao, WenWen Wang, XiaoLi Gong, and Jin Zhang. 2019. SELWasm: A Code Protection Mechanism for WebAssembly. In *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom).* IEEE, 1099–1106.

[18] Antero Taivalsaari and Tommi Mikkonen. 2018. On the development of IoT systems. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC).* IEEE, 13–19.

[19] Antero Taivalsaari, Tommi Mikkonen, and Kari Systä. 2014. Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. In *2014 IEEE 38th Annual Computer Software and Applications Conference.* IEEE, 338–343.

[20] Mark Weiser. 1991. The Computer for the 21 st Century. *Scientific american* 265, 3 (1991), 94–105.

[21] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring IoT up to Speed with A WebAssembly OS. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops).* IEEE, 1–4.

[22] World Wide Web Consortium. 2019. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-1/ https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.

[23] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion.* 301–312.