MSc thesis

Master's Programme in Computer Science

Towards Adaptive WebAssembly Applications: Leveraging Capabilities of the Execution Environment

Paulius Daubaris

May 25, 2021

Faculty of Science

University of Helsinki

Supervisor(s)

Prof. Tommi Mikkonen, Dr. Niko Mäkitalo

Examiner(s)

Prof. Tommi Mikkonen, Dr. Niko Mäkitalo

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Koulutusohjelma — Utbildningsprogram — Study programme Master's Programme in Computer Science

Tekijä — Författare — Author

Paulius Daubaris

Työn nimi — Arbetets titel — Title

Towards Adaptive WebAssembly Applications: Leveraging Capabilities of the Execution Environment

Ohjaajat — Handledare — Supervisors

Prof. Tommi Mikkonen, Dr. Niko Mäkitalo

Työn laji — Arbetets art — Level MSc thesis	Aika — Datum — Month and year May 25, 2021	Sivumäärä — Sidoantal — Number of pages 49 pages

Tiivistelmä — Referat — Abstract

Designing software for a variety of execution environments is a difficult task. This is due to a multitude of device-specific features that must be taken into account. Hence, it is often difficult to determine all the available features and produce a single piece of software covering the possible scenarios. Moreover, with varying resources available, monolithic applications are often hardly suitable and require to be modularized while still providing all the necessary features of the original application. By employing units of deployment, such as components, it is possible to retrieve required functionality on-demand, thus adapting to the environment. Adaptivity has been identified as one of the main enablers that allow leveraging offered capabilities while reducing the complexity related to software development.

In this thesis, we produced a proof-of-concept (PoC) implementation leveraging WebAssembly modules to assemble applications and adapt to a particular execution environment. Adaptation is driven by the information contained in metadata files. Modules are retrieved on-demand from one or more repositories based on the characteristics of the environment and integrated during execution using dynamic linking capabilities.

We evaluate the work by considering what is the impact of modular WebAssembly applications and compare them to standard monolithic WebAssembly applications. In particular, we investigate startup time, application execution time, and overhead introduced by the implementation. Finally, we examine the limitations of both, the used technology and the implementation, and provide ideas for future work.

ACM Computing Classification System (CCS)
General and reference → Document types → Surveys and overviews
Applied computing → Document management and text processing → Document management → Text editing

Avainsanat — Nyckelord — Keywords

WebAssembly, adaptivity, package management

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

With the increasing number of heterogeneous devices and their appliance in different paradigms, such as Internet of Things (IoT), software development becomes increasingly challenging [33]. Devices vary from simple systems-on-a-chip to powerful workstations and thus support different technologies, possess various capabilities, and provide contrasting amounts of resources. Such heterogeneity increases the complexity of software development and requires solutions that could alleviate these hardships.

Due to the diversity of devices, their capabilities, and varying execution context, adaptivity is necessary to properly leverage the execution environment [7]. It has been identified as the core capability suited to reduce the complexity of information systems [44]. To achieve adaptivity, it is common to divide applications into multiple building blocks each conforming to a specific functionality targeting a particular environment [7, 45, 17, 32]. Such a technique enables the reduction of the startup time when compared to monolithic applications. In addition, it empowers to assemble applications on-demand based on the characteristics of a device or a platform. Nevertheless, different platforms often rely on a limited number of software stacks incapable of running a plethora of heterogeneous applications [51]. Therefore, the development is bound to a limited number of programming languages, and using a variety of technologies at once is hardly an option.

As a result, we shift our focus to WebAssembly [10] – a relatively young, platform-independent technology, which serves as a compilation target that can be used by a numerous of programming languages. The technology has proved to be useful in different contexts, which include wearable devices [29], microcontrollers [25], and, most notably, serverless computing [27, 40, 36, 19, 48]. With the recent efforts to establish dynamic linking capabilities [37], WebAssembly applications can be modularized. Modularity creates the opportunity to assemble applications capable of adapting to different execution environments on-demand by taking advantage of the device and platform capabilities available during execution.

The main objective of the thesis is to implement a lightweight runtime capable of retrieving WebAssembly modules on-demand based on attributes characterizing the environment and integrating them during execution. Furthermore, we evaluate the system in terms of startup time and potential overhead it introduces by presenting the published results [37]. Moreover, we discuss the pitfalls and shortcomings of WebAssembly technology and implementation that would need to be addressed in future work.

The research is driven by three research questions that we seek to answer:

1. What mechanisms enable adapting on-demand during execution using WebAssembly?
2. How do modular WebAssembly applications compare to monolithic WebAssembly applications?
3. What are the current limitations of the technology and the designed system preventing to build a better solution?

As the research method, we used design science introduced by Hevner et al. [28]. Ultimately, design science is based on the construction of an artifact and its evaluation [28]. Typically, the process follows multiple iterations enabling to refine both the artifact and the derived results [28]. In this work, the research involved multiple iterations regarding requirement gathering, background analysis, implementation, and evaluation.

The thesis is structured as follows. In Chapter 2, we contextualize the proof-of-concept (PoC) implementation by investigating adaptive systems, package managers, WebAssembly, and relevant features that influenced the design decisions. Chapter 3 introduces the system based on the most significant properties. Chapter 4 presents the measurements comparing monolithic and modular WebAssembly applications answering the first research question. Chapter 5 provides answers to the remaining research questions and identifies the shortcomings of WebAssembly technology and the built system. Finally, Chapter 6 concludes the thesis and presents the final remarks.

2 Background

In this chapter, we provide information contextualizing the implementation. In Section 2.1, we investigate how adaptation is achieved by software that is intended to both, be executed in multiple execution environments and leverage available capabilities of the device. Section 2.2 considers package managers, which embrace code reuse and enable software assembly based on the user needs, thus adapting to particular demands. Moreover, we iterate through the concepts used by the implementation, such as repositories and metadata. Finally, Section 2.3 introduces WebAssembly technology, which provides the capability to write software in multiple programming languages and execute it on a multitude of execution environments.

2.1 Achieving Adaptivity

Adaptivity can be achieved by adjusting the system to particular needs using available variables or integrating new building blocks during execution [38, 39]. These two categories are distinguished as a parameter and compositional adaptation [38]. The former is limited to work with what is available to the system. Hence, the approach is incapable of introducing new external mechanisms that could handle a broader set of issues [38]. The latter, on the other hand, can integrate new constructs on-demand and extend the capabilities of a system [38]. Throughout the thesis, we will be focusing on compositional adaptation due to the nature of the built system.

It is certainly possible to attempt covering all the possible hardware- and software-specific scenarios in a single code base. *Fat binaries* can be considered as an example. Such binaries contain different types of processor architecture implementations [32, 31]. However, this particular approach does not scale well and bloats the object file with redundant instructions [18].

Instead of working on a single application covering as many scenarios as possible, a different approach is to split it into separate building blocks. The component-based software approach has been significant to mobile systems due to the possibility to assemble applications taking device characteristics into account [17, 45, 7].

However, manually assembling applications for multiple environments using components is unreasonable [45]. The increasing number of components correlates to the degree of software diversity and variability. As a result, complexity is unavoidable and poses threats to the integrity of the software [47]. Fortunately, adaptivity has been identified as one of the main enablers capable of alleviating these issues [44].

Assembling components in an automated manner creates the possibility to alternate between different component implementations providing essentially the same range of capabilities [7]. Belaramani et al. [7] distinguished such a technique as functionality adaptation, which employs components to compose software based on the context of execution and device characteristics. As a result, the software is modular and flexible as it does not need to make prior assumptions about the context in which it will be executed.

To relieve the complexity of mobile application development, Rosa and Lucena [45] introduced a solution enabling software to adapt to a particular mobile device based on the available features. Such an approach enables applications to be highly flexible and portable since device- and capability-specific functionality is decoupled from the application itself and can be assembled dynamically by selecting the most appropriate component available [45].

Another example of adaptive and portable component use, that leverages specific capabilities of the target environment has been introduced by Kicherer et al. [32]. In this particular work, the implementation leverages available processing unit (e.g., CPU, GPU) capabilities dynamically using libraries [32]. Accelerator-specific parts are decoupled from the implementation and can be selected flexibly – on-demand during execution [32].

The introduced approaches share common techniques to achieve adaptation to a particular execution environment. Instead of writing *ad hoc* solutions for each platform available, applications are divided into multiple building blocks, whether these are considered to be components or third-party libraries. The building blocks are selected and assembled during execution based on the information characterizing the execution environment.

Monolithic systems have been identified to have issues with difficult maintenance [15], extensibility and portability [12]. Therefore, nowadays, a common practice is to reuse and integrate different building blocks with the existing codebase to assemble the desired application. Software components enable software to be modular, reusable, and

Name	**Description**
precision	Number format
signature	Function signature
psize	Problem size
target	Type of processing unit
pmodel	Programming model
cpu_features	Implementation requirements
speed	Coarse performance

Table 2.1: List of attributes for describing components. Adapted from [32].

quickly developed [50]. Moreover, dividing applications into multiple units creates the opportunity to execute software on resource-constrained devices by selecting components based on the available resources [7].

Nonetheless, to determine whether a component is suitable to be used in the current context of the execution, it must be accompanied by a sufficient amount of information [21], also identified as the metadata. Usually, the format and placement of the metadata are non-uniform. For example, some solutions employ separate metadata files to describe the components [7, 17]. Others use metadata files to describe the whole application rather than individual components [45]. Kicherer et al. [32] proposed embedding component information into an Executable and Linking Format (ELF) shared library, which comprise data required to construct processes [30].

The information characterizing components also varies between different literature. In the article presented by Kicherer et al. [32], the libraries are described using attributes shown in Table 2.1. In this particular approach, the information provided by the attributes is performance-oriented [32]. To improve the building block selection even further, libraries are characterized using performance measurements performed locally [32]. As a result, it is possible to take real measurements into account and provide the most optimal library choice available [32].

The use of attributes also creates the opportunity for the applications to be ported to different platforms [32]. Nevertheless, the described list of attributes distinguished by the authors is not exhaustive and can be extended with custom requirements [32]. However, expanding the list of attributes should be considered as it might affect the component selection by setting unrealistic demands which cannot be fulfilled [32].

An alternative to the attribute lists has been investigated by Dastgeer and Kessler [11] using conditional composition, which is a data-driven approach to component selection. The component selection is influenced by four different sources, which involve detailed information about the available resources of the system, platform-specific details, information related to the currently running application requiring the components (e.g., resource consumption) and information related to the actual component [11]. Fine-grained information enables the execution environment to take these sources into account and leverage components most appropriate for a particular context [11].

Components can also be accompanied by metadata files to provide context about its properties [17]. Fjellheim [17] proposed using three different properties to contextualize a component: *activity*, *java platform* and *code size* [17]. However, the presented information is not considered to be finite [17]. If needed, the information used to determine if a component is suitable for a particular environment can be extended with more descriptive properties, such as device features [17].

Belaramani et al. [7] introduced facets, which comprise a small, stateless component and its description represented by a single XML file [7]. The metadata of the component has a fixed number of properties. Facets are distinguished using an *id* alongside the *funcID* identifying its capability [7]. Furthermore, metadata includes facet creator name [7]. A single facet can have multiple implementations and thus be represented by multiple versions [7]. Moreover, due to resource constraints on mobile devices, metadata includes resource requirements of a facet [7]. Lastly, the metadata includes information about additional dependencies a facet can rely on [7].

Most of the component descriptions employ information of different granularity. However, they do share common properties. For example, most of the investigated approaches identify the functionality a component provides. This does not apply to accelerator-specific features introduced in [32, 11] because the functionality provided by the components is already known. Because a component can depend on multiple components, dependency information is also popular among different concepts. However, in performance-oriented cases, rather specialized information is used. For example, in the work presented by Kicherer et al. [32], the authors employ a separate performance database to select a component, which would determine the most efficient library in comparison to other available choices. Similarly, for resource-constrained devices, Fjellheim [17] introduces a property to identify code size to determine whether a component is suitable for a particular execution environment.

The introduced systems employ components and additional metadata to adapt to different needs. Some approaches tackle adaptation to a particular execution environment taking its characteristics into account, while others attempt to achieve optimal performance. Usually, the inspected systems avoid making assumptions about the environment before the execution and retrieve components on-demand based on information available during execution. The components are described by a data-driven technique, such as using a separate metadata file characterizing the component or information embedded into a binary. Furthermore, the building blocks are usually injected into a running application using techniques, such as dynamic linking.

2.2 Package Management

Package management reduces the complexity of software development by providing the capability to use already written code offering varying functionality [41]. Package managers usually involve a collection of mechanisms including package retrieval from remote repositories or mirrors, their integration, upgrading to specific versions and removal [3, 49]. A significant capability of package management is to relieve developers from manually handling software on which a package depends and instead, solving the problem in an automated manner thus preventing a well-known issue of dependency hell [49]. Nevertheless, package management features are subjective. There is no de facto standard describing the requirements of such a system. As a result, different package management solutions had emerged throughout the years with contrasting features. For example, Abate et al. [2] proposed a modular package manager capable of alternating between different dependency solving mechanisms. Brown et al. [8] expressed concerns regarding the security of numerous package managers and thus proposed additional mechanisms providing the capability to mitigate some of the most pressing issues.

Depending on the nature of a package manager, it can provide software in different forms. Systems encountered in many Linux distributions usually use pre-built packages and settle upon convenience [6]. However, pre-built packages fall short in terms of unconventional execution environments possessing contrasting capabilities [6]. Nevertheless, not all package managers use pre-compiled binaries. A particularly flexible package manager (also identified as a port system [20]) is Portage, Gentoo package manager [23]. In Portage, a package is considered to be a set of instructions required for software installation [4, 13]. This means that the process of compilation is actually performed locally.

A unique aspect of Portage is its capability to adapt the software to a particular environment and user requirements. Such mechanism is achieved by using *USE* flags [22]. *USE* flags influence the compilation process and impact the produced software by the user explicitly stating the desired properties that an application should include [4]. For example, instead of compiling a complete software package, a user can select a subset of optional features, thus adopting a lighter version of the desired piece of software [4].

Using Portage, the software can be adapted to different environments and adjusted based on user preferences by providing fine-grained control over the available build options. However, such flexibility comes at a price of the performance and even the possibility of the desired package being impossible to build [13]. Furthermore, since the software compilation is performed on the client-side, this does not scale well on resource-constrained devices [6]. As a result, Bal-Pétré et al. [6] proposed a different approach that takes advantage of the Portage package manager capabilities. Instead of building applications locally, the authors proposed using cloud capabilities and alleviating the execution environment from a consuming task of compilation [6].

Although package management features vary between implementations, remote storage is an essential part of package management. Package managers communicate with repositories to download the necessary building blocks [3]. Most package repositories offer the capability to store and retrieve available components. Besides such component storage, package metadata is another important component of package management. Metadata characterizes a package and usually comprises similar information between different package managers. However, the actual placement of the metadata is non-uniform.

Linux distributions typically use an arbitrary archive format to encompass the files of a package with additional information about the package itself embedded into the specified format [9]. Package managers, such as Node package manager [42] (*npm*) store package specific information in an external *package.json* file [60]. An interesting aspect of the metadata is that it supports the use of the abbreviated form of metadata. It can accept fewer configuration options compared with a full metadata format [43]. Metadata of *npm* provides the capability to describe the software for development and production environments [60]. It is achieved by using the metadata properties –

dependencies and *devDependencies* [60]. By using such dependency categories, it is technically possible to adapt to two different contexts.

Packages often depend on additional third-party code called dependencies [9]. To install the actual package and its prerequisite code, package managers employ dependency solvers [3]. Usually, packages are denoted by information relevant to the dependency solver, for example, the version of a package. However, versioning complicates package installation especially if a package manager does not provide the capability to leverage different variants of the same dependency [1]. As a result, dependency solving is identified as an NP-complete problem [2, 1]. However, the recent results published by Abate et al. [1] show that the current dependency solving solutions employed by package managers are improving due to the usage of well-known algorithms (e.g., MaxSAT).

Package managers are essential to managing software and its dependencies. Such systems are often specialized for different execution environments or programming language ecosystems. The resources offered by the repository can be represented in different forms, such as pre-compiled packages, source code, or compilation instructions providing the information regarding arbitrary software installation. Packages are accompanied by metadata, which can be embedded into an archive format or introduced as a separate file. Lastly, to correctly install packages, package managers employ dependency solving mechanisms.

2.3 WebAssembly

WebAssembly is a low-level code format, which can be used as a compilation target by numerous programming languages and deployed to a multitude of environments [10, 26, 59]. Such format exceptionally increases flexibility as applications can be written in a variety of languages alleviating the restrictions of choosing a particular technology stack for software development [10]. Moreover, WebAssembly is secure. The security of the technology is enforced by executing the code within a virtual machine (VM) disallowing any communication between the execution environment unless specified otherwise by the embedder (e.g., an operating system) [10]. The virtualization aspect of WebAssembly is significantly lighter than those of the traditional OS-level virtual machines or containers [36]. Such characteristics enable the format to be used in a variety of environments where speed, security, and portability matter. A few examples

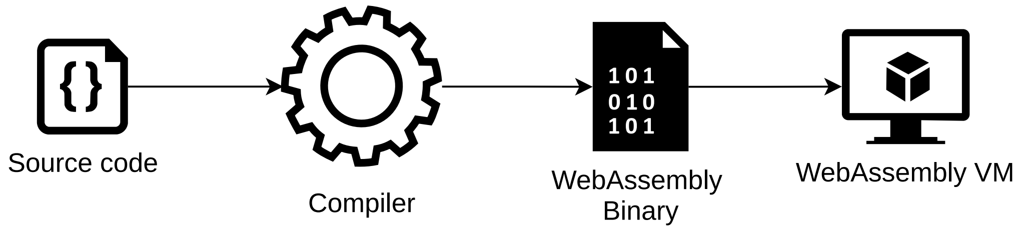

Figure 2.1: Overview of how WebAssembly is used.

of such environments are Internet of Things (IoT) or cloud and edge devices. An example of steps involved to achieve the execution of a WebAssembly program from source code is shown in Figure 2.1. Source code can be written in a variety of programming languages (e.g., C, C++, Rust). The source code is processed by a compiler, which emits a WebAssembly binary. Finally, the binary can be executed on a device that contains a WebAssembly Virtual Machine (VM).

To execute WebAssembly applications, a Virtual Machine is required. Wen and Weber [59] describe WebAssembly VM as a ”program that translates WebAssembly binary instructions to native CPU machine codes before execution”. A VM might involve a variety of different types of compilers to translate binary instructions, among which, the most notable are streaming compilers. For example, browsers, such as Google Chrome, usually use two-tiered compilation, which involves employing two different compilers [52]. The first compiler is responsible for generating code as fast as possible, usually compiling at the same time the chunks of code are retrieved over the network, and the other compiler re-compiles the code and optimizes it [52].

To date, there are approximately 30 open-source standalone WebAssembly VMs developed [5]. The VMs vary by their implementation, and not all of them conform to the test suite of the technology or are not actively maintained by the open-source community. Moreover, some available VMs are developed by researchers or by major companies. For example, Wasmtime VM[*] is developed by developers of the Bytecode Alliance project[†], which involves companies, such as Fastly, Intel and others.

A WebAssembly VM can be the deciding factor in the performance and portability of WebAssembly applications [61]. Different VMs offer contrasting features which influence the capabilities of the application. For example, some VMs implement features that are not yet standardized, while others strictly follow the specification and intro-

[*]https://github.com/bytecodealliance/wasmtime accessed November 21, 2020.
[†]https://bytecodealliance.org accessed November 21, 2020.

duce new features only when these are standardized.

Even though WebAssembly applications run in a sandboxed environment, the capabilities can be expanded with the use of host functions. A host function is a function defined by the environment in which the application is intended to be executed and supplied to a module as an import [10]. Besides the capability to access numerous resources of the execution environment, host functions can also affect the state of the application [26]. An example has been shown by Watt [54], where host functions were leveraged to manipulate the heap memory.

Host functions can be considered as the enabling feature of the work presented in the thesis. Host functions are used to modify and grow memory dynamically and perform symbol relocations. Moreover, with the capability to exit the sandbox, we leverage the resources of the execution environment, which involve access to the network and the filesystem. Ultimately, host functions enable us to achieve the implemented system.

Among host functions, memory, and table components play a significant part in the implementation. Memory in WebAssembly is linear, meaning that it is a contiguous block of bytes that can be imported by different modules in case it is exported [10]. A table is a similar component to memory, however, it differs in the information it is designed to represent [10]. As of WebAssembly version 1.1, a table is a vector capable of holding function references [10]. If a table is shared between modules, it empowers a module to invoke other module functions. Currently, WebAssembly modules are limited to using only one memory and table [10]. As a result, if a module exports these components, all the stored information will be fully available to other modules if imported.

Nonetheless, the multiple memory proposal [56] has been created to address some of the issues that the current memory model possesses. For example, it could provide a mechanism to create multiple memories which could also be separated into public and private memories, thus preventing access to data that was not intended to be shared [56].

For portability reasons, memory is grown by using a fixed size of 64 KiB, or in other words – a page [10]. However, fixed-size memory growth, specified by the specification, has posed issues when executing WebAssembly applications on microcontrollers [61]. As a result, the specification might need to be adjusted to be a feasible choice for environments offering a limited amount of resources.

ID	Section	ID	Section	ID	Section
0	Custom Section	4	Table section	8	Start section
1	Type section	5	Memory section	9	Element section
2	Import section	6	Global section	10	Code section
3	Function section	7	Export section	11	Data section

Table 2.2: Sections of a WebAssembly binary. Source [10].

Besides storing the data, memory can be used for more sophisticated data representation in case the offered data types are too restrictive. Currently, the technology enables the usage of only four different data types, which include 32-bit, 64-bit integers, and 32-bit, 64-bit floating-point representations [10]. However, with the use of memory, it is possible to leverage more complex types, such as null-terminated character sequences. Nevertheless, a 32-bit integer is rather significant as it can be used as a pointer [46, 10] and thus access a specific memory region or refer to a function stored in a table [26].

In combination with host functions, it is possible to read and modify the memory of a module via the host environment and copy data to other available modules in case these do not share the same linear memory. Such actions can be achieved by using a WebAssembly VM embedding application programming interface (API). The API is available to the host environment and can be leveraged by host functions.

Because WebAssembly modules can be written in a variety of programming languages, some of them might possess different application binary interface (ABI). As a result, data types, such as a string, might be represented in a contrasting form. Thus, for a WebAssembly program to behave correctly, additional care needs to be taken into account when working with more sophisticated data types.

Lastly, WebAssembly modules are represented by a set of sections shown in Table 2.2. Most of the sections must comply with the requirements regarding their content and placement in a binary imposed by the specification [10]. Nevertheless, custom sections are less constrained. The content of a custom section can be freely defined by the developer. Moreover, custom sections are not restricted by the specification in comparison to the remaining sections and can be used multiple times and placed in-between any other section [10]. However, using the information provided by custom sections requires knowledge of the embedded data and an understanding of how to decode it. Thus additional application logic is required to leverage the embedded information.

3 Implementation

In this chapter, we describe the implementation, which builds upon our previous work [37]. Section 3.1 describes the core features of the implementation in relationship to the background information. In Section 3.2, we provide a detailed overview of the created system, which highlights the most important features. Section 3.3 details the adaptivity enabling features. Specifically, we describe WebAssembly modules, which serve as building blocks of the application, the concept of interfaces, and attributes characterizing the modules. Lastly, Section 3.4 includes information about how WebAssembly modules are retrieved, selected, and integrated by the module management mechanism.

3.1 Motivation and Features

The goal of the implementation was to reduce software development complexity for heterogeneous devices and leverage the capabilities of the execution environment without rewriting the entire application. We use WebAssembly to create modules isolating capability-specific features and assemble applications on-demand based on the characteristics of the environment capable of executing in different contexts.

The implementation takes inspiration from two main concepts. Firstly, we capture ideas from adaptive systems, such as using modules as building blocks to compose the desired applications and their descriptions to describe the suitability of a component for a particular execution environment. Modularity creates the opportunity to isolate capability-specific functionality and maintain the core of the application-independent from redundant code. We use attributes to describe the capabilities of both the execution environment and modules. Module functionality is identified by an interface allowing to find the most suitable implementation available.

Secondly, we use ideas widely reflected in package managers, including repositories serving as module storage. Nonetheless, we assign additional responsibilities to the repository, such as performing the module selection process based on the characteristics of the environment. Another difference between the implementation and usually encountered package managers is the capability to retrieve building blocks during ex-

Name	**Description**
dlopen	A host function responsible for opening and loading a WebAssembly during execution.
dlsym	A host function, which resolves functions from loaded modules.

Table 3.1: Host functions enabling to integrate and use functionality exposed by WebAssembly modules.

ecution. Furthermore, we employ metadata files to embed the previously mentioned information describing a module and its dependencies. Finally, we integrate modules using dynamic linking.

3.2 Overview of the Implementation

Figure 3.1 represents a detailed overview of the implementation, starting with the application invocation and proceeding with the action to activate the camera, which is accessed via functionality offered by a separate module retrieved from the repository. The implementation is mainly driven by two host functions, *dlopen* and *dlsym*, shown in Table 3.1 and described in more detail later in the chapter.

The application starts on a laptop that has an integrated camera. Application begins its execution by requesting the module A. The request is handled by the module manager library, which initiates the search by firstly investigating the filesystem. Because the module is not found locally, the module manager creates a request to the repository. The request includes the interface of the desired functionality and characteristics of the execution environment. The repository location is determined based on an initial metadata file, which in this figure is considered to be pre-installed. Whenever the repository receives the request, the module is selected based on the received information and returned to the runtime. The initial module A is then compiled to native code using the WebAssembly VM embedding API, which involves actions, such as supplying the required imports, compilation, and instantiation. Both memory and table components will be supplied to the remaining modules. Therefore, the information and functionality will be accessible to all the integrated modules.

When module A is integrated, the execution follows the user activating the camera. However, with camera functionality not being included, module A proceeds to call *dlopen* and issue a request to the repository for module B, which encapsulates the needed functionality to take advantage of the camera. As before, the request is sent containing the environment and the module characterizing information. In this case, however, the repository searches for the metadata of the module because it can possess additional dependency information. Upon retrieving the metadata of module B, the metadata is analyzed. Since there no additional dependencies were found, another request is issued directly to the repository listed in the metadata of module B. In addition to the same steps introduced earlier, the retrieved module B is supplied with the global memory and table components of the initial module. When the module is integrated, the user can resolve the functions using *dlsym* and use the camera.

3.3 Adaptive Features

The implemented system utilizes WebAssembly modules as building blocks of the application. Modules need to be compiled with a specific set of instructions to produce data required by the implementation. Furthermore, modules are characterized by information specific to a particular execution environment, which influences the module selection mechanism.

WebAssembly Modules

WebAssembly modules are created by using a compiler capable of emitting WebAssembly code. A commonly used toolchain capable of targeting WebAssembly is Emscripten*. Emscripten uses the LLVM backend to create WebAssembly binaries. The compiler frontend is capable of producing two different types of standalone WebAssembly modules – main and side modules. The difference between the main and side module is that the main module has direct access to system libraries as opposed to the side module [16]. Nevertheless, in case a side module requires a specific system library not required by the main module, it is possible to either produce the main module with all the available system libraries or to determine, which libraries are required by side modules. This is achieved by explicitly listing the libraries when building the main module so that the main module would include only what is necessary [16].

The implementation requires WebAssembly modules to carry information used for the integration because it is generally unknown, for example, how much memory a system

*https://emscripten.org/index.html, accessed March 10, 2021.

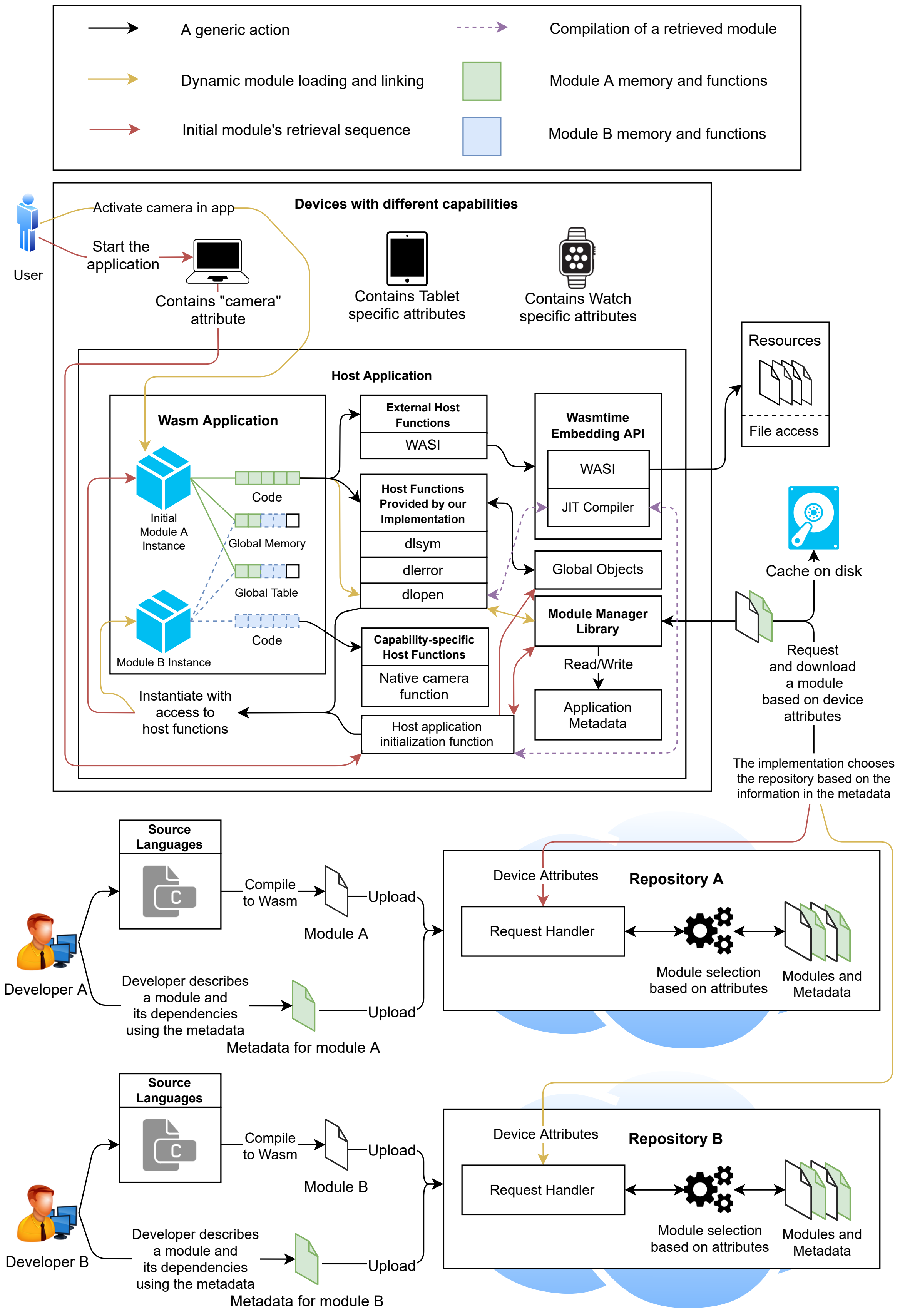
A generic action
Compilation of a retrieved module
Dynamic module loading and linking
Module A memory and functions
Initial module's retrieval sequence
Module B memory and functions
User
Activate camera in app
Start the application
Devices with different capabilities
Contains "camera" attribute
Contains Tablet specific attributes
Contains Watch specific attributes
Host Application
Wasm Application
Initial Module A Instance
Code
Global Memory
Global Table
Module B Instance
Instantiate with access to host functions
External Host Functions
WASI
Host Functions Provided by our Implementation
dlsym
dlerror
dlopen
Capability-specific Host Functions
Native camera function
Host application initialization function
Wasmtime Embedding API
WASI
JIT Compiler
Global Objects
Module Manager Library
Read/Write
Application Metadata
Resources
File access
Cache on disk
Request and download a module based on device attributes
The implementation chooses the repository based on the information in the metadata
Source Languages
Compile to Wasm
Module A
Upload
Developer A
Developer describes a module and its dependencies using the metadata
Metadata for module A
Device Attributes
Repository A
Request Handler
Module selection based on attributes
Modules and Metadata
Source Languages
Compile to Wasm
Module B
Upload
Developer B
Developer describes a module and its dependencies using the metadata
Metadata for module B
Device Attributes
Repository B
Request Handler
Module selection based on attributes
Modules and Metadata

Figure 3.1: Detailed overview of the implementation.

Field	**Type**	**Description**
memorysize	varuint32	Size of the memory area the loader should reserve for the module.
memoryalignment	varuint32	The required alignment of the memory area.
tablesize	varuint32	Size of the table area the loader should reserve for the module.
tablealignment	varuint32	The required alignment of the table area.

Table 3.2: The *dylink* custom section. Adapted from [58].

should allocate for a particular module. As a result, each module that can be integrated holds information illustrated in Table 3.2. The information is embedded into a custom section of a WebAssembly module and represented by *dylink* name, which indicates a dynamic library [58]. As a result, if the custom section is not present within a WebAssembly binary, the module can be discarded as it is unfit to be used with the implementation.

By default, a WebAssembly module is executed in isolation [10]. This means that module components, such as memories and tables, can only refer to the definitions within the bounds of a module. Nevertheless, the implementation breaks the set boundaries to achieve benefits, such as direct data and functionality sharing. Instead of each module having its memory, the runtime maintains a contiguous block of memory. For a module to distinguish its memory, its location is identified by *__memory_base* variable that is provided to the module when it is instantiated and allows it to locate the static memory region it owns. The same principle applies to the table component – each table is identified by a *__table_base* variable. The runtime maintains a global table that holds all the function references defined by different modules. Any module can access any function defined by any module.

Module Characterization

The device that runs the application can be characterized by a set of attributes, such as architecture, operating system, and features that are available on the device (e.g., camera). The same principle applies to modules, which use attributes to describe what characteristics a module supports.

The attributes are used to influence the module retrieval process, which enables the module selection mechanism of leveraging not only platform-, but also device-specific capabilities. In practice, this means that an application can automatically adapt to its execution environment.

However, at the current state of the implementation, we do not provide an exhaustive list of attributes that could be used. This would require an agreement on what are the available options and what capabilities could be leveraged among different devices. For now, the PoC allows any type of attribute to be used. However, as mentioned by Kicherer et al. [32], such flexibility imposes the risk of being incapable of finding a suitable module. Nonetheless, such issues are considered to be addressed in future work.

Furthermore, we use an abstraction of an interface. An interface is used to define specific functionality that several modules can provide. Modules conforming to the same interface can implement functions in different ways and thus provide flexibility through the use of a module best fit for the given situation. An interface is identified by a unique ID.

Lastly, a module can depend on multiple modules. Thus we enable to define dependencies, which are separated into two categories: load-time and execution-time. The former needs to be loaded before the desired module because it relies on dependency functionality. The latter, on the other hand, can be loaded on-demand, whenever they are needed.

3.4 Module Management

The built system takes advantage of some of the package management concepts described in Section 2.2. In particular, we employ software repositories, which in addition to serving as storage for WebAssembly modules, exploit a mechanism that enables finding suitable modules conforming to the characteristics of the execution environment. Moreover, the module management is based on the metadata information, which includes previously introduced information characterizing a module.

Module Metadata

Each WebAssembly module is identified and characterized by a separate metadata JavaScript Object Notation (JSON) file. The dedicated metadata file maintains information, such as an interface ID to which a module conforms, the location at which the module resides, attributes describing capabilities of a module, and, finally, top-level dependencies that it requires. Internally, the metadata is represented by the *Metadata* structure displayed in the Listing 3.1. An interface ID does not represent a single module, but rather multiple modules that provide the same functionality. The use of interfaces in the implementation enables to provide multiple options to choose from when retrieving a module for a device. The selection is ultimately based on the attributes and the interface ID.

```
struct Metadata {
    id: String,
    attributes: Vec<String>,
    location: String,
    dependencies: HashMap<String, Dependency>,
}
```

Listing 3.1: Structure representing a metadata file.

The dependencies are represented by a hash map data structure where key-value corresponds to the interface name of a dependency and value to the information described by the *Dependency* structure shown in Listing 3.2. The structure encompasses information, such as an identification string to distinguish a module, a flag indicating if a module is a load-time or an execution-time dependency, and, finally, the location at which the dependency resides.

```
struct Dependency {
    id: String,
    load_time: bool,
    location: String,
}
```

Listing 3.2: Structure representing dependency information.

An example of a metadata file in JSON is shown in Listing 3.3. The file describes a module conforming to the interface ID *camera* which requires the platform to support the *linux* platform and is placed in a local server by the name *v4l2.wasm*. The dependency object contains information about packages that a module depends on. In the case of the shown example, the *camera* module depends on a *color* and *grayscale* filters. The main difference between the given dependencies (besides providing different functionality) is the moment at which they are required to be loaded. In essence, *load_time* keyword indicates two cases at which a dependency must be loaded. Firstly, if the flag is set to *true*, then the dependency must be loaded before the desired module. Otherwise, the dependency can be deferred to be loaded at a later time – at which it is requested.

```
{
    "id": "camera",
    "attributes": ["linux"],
    "location": "http://localhost:8000/v4l2.wasm",
    "dependencies": {
        "color_filter": {
            "id": "color_filter",
            "load_time": false,
            "location": "http://localhost:8000"
        },
        "grayscale_filter": {
            "id": "grayscale_filter",
            "load_time": true,
            "location": "http://localhost:8000"
        }
    }
}
```

Listing 3.3: Metadata of an example module.

Requesting a Module

Whenever a WebAssembly application issues a request for a specific module using the provided API, the runtime will firstly inspect if that particular instance already exists. If the module has already been instantiated and found in the global array of instance information, it will be used and the search for it elsewhere will terminate. Otherwise, the search continues beyond the memory of the program.

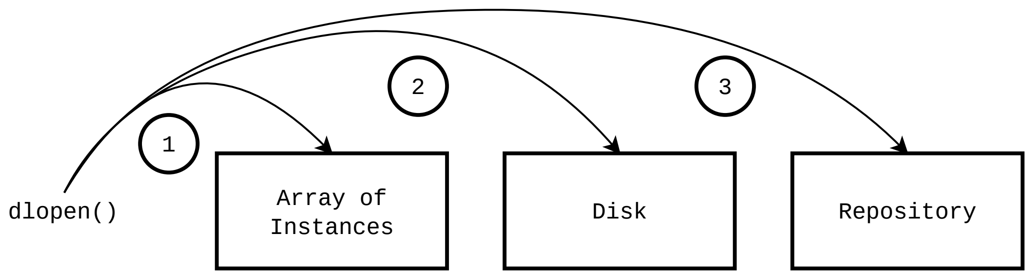

Figure 3.2: Search order of a module.

The runtime search process is similar to Linux dynamic linker in a way that it defines multiple locations from which a module can be retrieved from [30]. The implementation prioritizes the search for a required module locally to minimize the expected delay, which can be caused by the subsequent retrieval mechanisms invoked in case a module is not found. In case the module is not already integrated, a predetermined *cache* location is investigated. The location is dictated by the initial metadata file. If the module is not found locally, it will issue a request to the repository where it is expected to reside. The order of searching for a module is illustrated in Figure 3.2.

The first two search procedures are straightforward, however, retrieving a module from a repository is not as direct. A separate module management library was created to provide the ability to retrieve WebAssembly modules from a remote repository. The library provides only two public methods: *new* and *load*. The *new* method creates an instance of *Manager* structure shown in Listing 3.4. Upon instantiation, the method searches for the initial metadata in the location provided by the argument supplied to the method. If the metadata file is not found, the execution will halt as the information stored in it is crucial for the implementation to perform correctly. Otherwise, the file is read and deserialized into a *Metadata* structure. Before finishing the instantiation process, the last process performed by the method is checking the cache location for existing modules.

Throughout the execution of the application, the *Manager* structure will populate its *metadata* field with information loaded from the module metadata. This allows to keep the information in memory and reduce the number of redundant calls to the filesystem or a repository. Nevertheless, upon the termination of the runtime, the accumulated metadata information would be lost.

The purpose of the *load* function is to retrieve required modules at any point during the execution. The process of loading the desired module includes additional processes,

```
struct Manager {
    metadata: HashMap<String, Metadata>,
    attributes: Vec<String>,
    cache: Option<String>,
    modules: Vec<String>,
}
```

Listing 3.4: The structure representing module manager.

such as loading its dependencies. The *load* function is invoked by the *dlopen* host function if the desired module is not already loaded. Upon invocation, the process will start by analyzing the metadata and collecting information about modules that the requested module depends on.

The full process is illustrated in Figure 3.3. The function *load* accepts the requested module's information described in the dependency section of the caller module's metadata. The process continues by retrieving metadata of the requested module and analyzing its dependencies. In case a dependency is identified as load-time, the module manager proceeds to issue requests to the repository to retrieve the load-time dependency metadata. The module manager continues by analyzing whether these include load-time dependencies and resolving them recursively until no load-time dependencies remain unresolved. Whenever this process is complete, the metadata of the load-time dependencies is stored in memory. Using the acquired information, the module manager retrieves the actual binaries to be linked. After these steps are complete, both the metadata and the modules are returned for further processing. The module manager is not responsible for the compilation or instantiation of a module. Instead, the module manager offers functionality, such as to download a module, resolve its dependencies, and cache it for subsequent use.

Module Selection

A module repository is expected to utilize a module selection mechanism based on the interface ID and the attributes, which are encapsulated in an issued request. The repository is not restricted to any kind of mechanism as long as it is capable of sending the correct modules back to the application. To clarify, a mock repository was

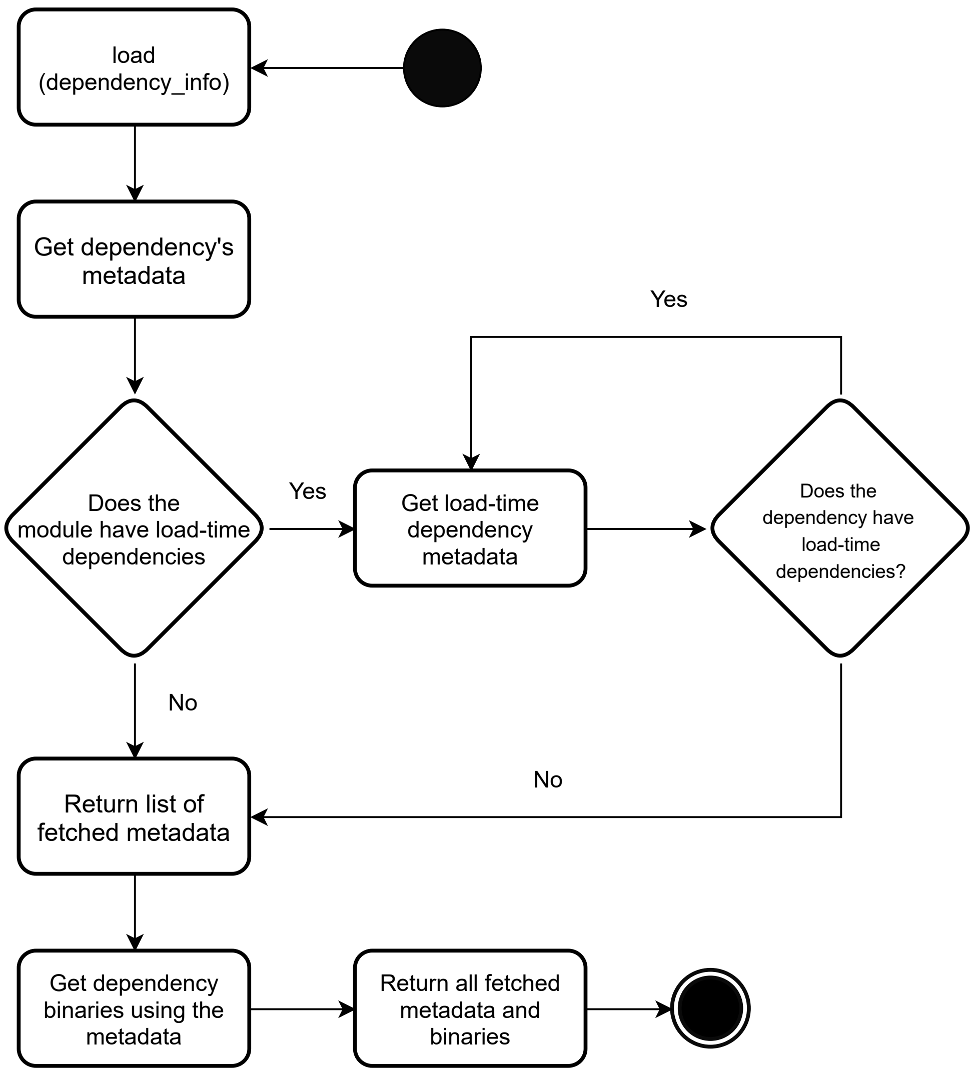
load (dependency_info)
Get dependency's metadata
Does the module have load-time dependencies
Yes
No
Get load-time dependency metadata
Does the dependency have load-time dependencies?
Yes
No
Return list of fetched metadata
Get dependency binaries using the metadata
Return all fetched metadata and binaries

Figure 3.3: Dynamically loading a module and its load-time dependencies.

created to illustrate the concept behind module selection using express*, Node.js web application framework.

Upon receiving a request, the mock repository finds all the metadata files conforming to the received interface ID. Next, it discards the metadata files which require attributes not listed by the query. Finally, the repository selects the best match from the aggregated metadata files – a metadata file supporting the most attributes and corresponding to the correct interface ID. When retrieved and analyzed, a module is downloaded directly by using the link provided in the metadata. A high-level overview of the module selection process is shown in Figure 3.4. The application is executed in an execution environment supporting two attributes: *camera* and *linux*. Because a

*https://expressjs.com/ accessed January 29, 2021.

module has not been found locally, a request to the repository is issued for the module conforming to the interface ID *color_filter* with a supported array of attributes included in the payload of the request. Upon receiving the request, the request handler filters out all the module metadata satisfying the requirement of the received interface ID. In the case of the illustration, the repository finds three metadata files conforming to the received interface ID. After analyzing the supported attributes of each metadata file, *color_filter_linux.json* is selected as it supports both of the attributes received. When returned, the metadata is ready to be analyzed and the module integrated.

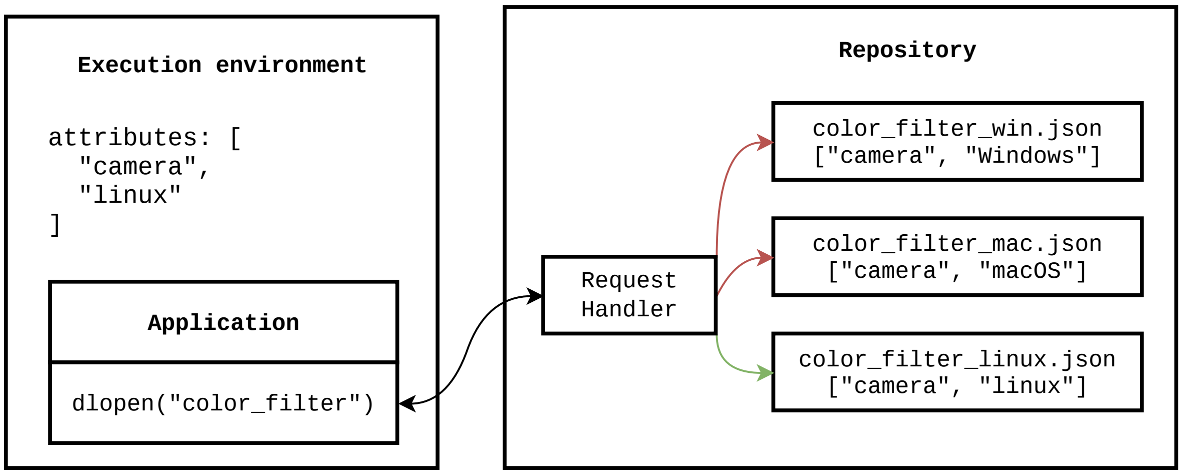
Execution environment
attributes: ["camera", "linux"]
Application
dlopen("color_filter")
Repository
Request Handler
color_filter_win.json ["camera", "Windows"]
color_filter_mac.json ["camera", "macOS"]
color_filter_linux.json ["camera", "linux"]

Figure 3.4: Module selection process.

Module Integration

For a module to offer its functionality, it firstly needs to be integrated into the application. This is taken care of by the previously introduced host functions listed in Table 3.1. Firstly, the runtime checks if a module can be dynamically linked – it verifies and analyzes whether the binary maintains the *dylink* section holding information necessary for the runtime to correctly allocate needed resources. The decoding is done using an internal function *check_dylink* provided by the runtime. The function decodes information encoded in Little Endian Base 128 (LEB128) encoding. The decoded information is represented by the *Dylink* structure shown in Listing 3.5.

Upon module retrieval, it is linked into the application with its load-time dependencies. The process of linking a module is composed out of multiple steps. Firstly, the runtime uses the information acquired from the decoded *dylink* section of a module. It allocates memory using the *malloc* function exported by the main module which is the original

```
struct Dylink {
    mem_size: u32,
    mem_align: u32,
    table_size: u32,
    table_align: u32,
    needed_dynlibs: Vec<String>,
}
```

Listing 3.5: The structure of the *dylink* section.

owner of the global memory and table objects and grows it by using the embedding API of the Wasmtime VM. The memory and table sizes are grown by *mem_size* and *table_size*, respectively.

Secondly, after all the allocation procedures are done, the module is compiled using the embedding API and is ready to receive imports, which are resolved by the Linker structure. The Linker is a convenience structure automating the process of supplying the correct imports required by a module [53]. When all of the imports are resolved, a module is instantiated utilizing embedding API after which the relocations are applied using *__post_instantiate* function provided by the instance of a module.

After a module is integrated into the executing application, the application can use its offered functionality by firstly resolving functions with the *dlsym* host function and then invoking them as regular functions. The host function is capable of resolving any function of any module integrated into the application. When invoked, *dlsym* searches for the requested function by name supplied to the host function. Afterward, the global table is grown and the function reference is inserted into the global table. Upon subsequent calls to resolve the same function by other modules, the function will be a lot faster to resolve since it has been placed inside a global table to which *dlsym* refers before searching through existing module instances.

The presented implementation is a combination of concepts. We apply practices of adaptive systems, such as the use of component-based software and descriptions accompanying building blocks. Similarly, we employ package management ideas, including metadata files, which embed information characterizing a module, and repositories, which, in addition to storing the components, enable their selection. The integration of modules is achieved by using dynamic linking.

4 Evaluation

In this chapter, we present the evaluation of the built system, which was introduced in our previous work [37]. We aimed to investigate the significance of modular WebAssembly applications and compare them to monolithic WebAssembly applications. In Section 4.1 we detail the setup that was used to perform the measurements. In Section 4.2 we proceed to introduce the startup time measurements. Following the startup time measurements, in Section 4.3, we measured the execution time. In Section 4.4 we discuss the overhead introduced by the implementation and finally, Section 4.5 analyzes the derived results.

4.1 Setup

The measurements were performed on 64 bit Ubuntu 18.04.5 LTS OS with 16 GB RAM and an Intel i5-7200U@2.50GHz x 4 processor [37]. WebAssembly modules were created by compiling applications written in C programming language to WebAssembly using Emscripten[†] compiler toolchain of version 2.0.0. Table 4.1 describes the used compilation flags required by WebAssembly lld port [35] to link a module and be compatible with the implementation. Similarly, Table 4.2 introduces compilation build options specific to Emscripten used alongside the compilation flags.

Flag	**Description**
-*Wl*	Indicates the arguments supplied to the linker.
–*import-table*	Enables a module to import a table. In the case of our implementation, a module imports a global table.
–*no-entry*	Indicates to omit the search of the entry point function.
–*allow-undefined*	Allows to have undefined symbols in a module.

Table 4.1: Compilation flags required to create a WebAssembly module compatible with our implementation. Source [35].

[†]https://emscripten.org/index.html accessed January 22, 2021.

Build Option	**Description**
-s ALLOW_MEMORY_GROWTH=1	Enables to change the memory size dynamically during execution.
-s ERROR_ON_UNDEFINED_SYMBOLS=0	Does not halt compilation or linking process in case an undefined symbol is encountered.

Table 4.2: Build options required by Emscripten to create a WebAssembly module compatible with our implementation.

4.2 Startup Time

The implementation enables applications to be split into multiple modules containing decoupled functionality. Instead of loading the whole application at once, with the modular approach, the application can be assembled gradually retrieving parts only when necessary. This implies that we reduce redundant functionality and employ only what is necessary at the present moment. As a result, the startup time can be decreased significantly. Also, it is important to note that the startup time measurement depends on how finely the application is divided. In this section, we compare the startup time imposed by both monolithic and modular approaches. The modular approach employs fine-grained modules, which are assembled on-demand. The measurement presents the impact on the time required to start the application up until the point at which it is ready to proceed with the execution. In addition, we present how much memory both processes require.

The startup time results are shown in Table 4.3. In essence, both applications are the same when fully loaded. A monolithic application is robust and incapable of introducing specific functionality during the execution. Moreover, it is slower to start, however, it does not require to load additional functionality. The modular approach, on the other hand, acts as a bootloader and comprises only the code required to start the application. As a result, the time required to start the application and both binary sizes are significantly different. Similarly, with the modular approach, the startup process requires a lot less memory during the early execution. This is particularly relevant to resource-constrained devices, which often cannot offer the luxury of storing the whole application at once at must compromise by either using the application in parts or referring to more powerful devices enabling to perform computations remotely. Nev-

Approach	Binary Size	Startup Time	RAM
Monolithic	15200 KB	4.472s	321284 KB
Modular	28.6 KB	0.104s	9852 KB

Table 4.3: Startup time measurements. Source [37].

ertheless, for the modular application, memory consumption increases when modules are loaded into the application throughout the execution.

4.3 Application Execution

The entire execution is measured similarly when compared to the startup time measurement. Here, however, we consider invoking a function and present different scenarios, which illustrate both the benefits and the weaknesses of the proposed modular application assembly. As before, the applications are equal when loaded completely. However, in the modular approach, the application is divided into relatively small modules representing a portion of the monolithic application.

In this measurement, we also consider the host function responsible for resolving functions because to use the functionality offered by a module, its functions firstly need to be resolved. Table 4.4 represents multiple modular application cases considered for the entire execution measurement. We do not consider additional configurations for the monolithic approach because it is not as flexible and all the functionality needs to be loaded at once. For the modular application, we consider cases where we load all the functionality and resolve either a single function or all available functions and where we load only a single module and resolve only the function required. These scenarios enable us to inspect the benefits of the modular approach and see how much overhead we introduce when loading all the functionality in comparison to the monolithic application. As before, the measurements can vary depending on module granularity.

Abbreviation	Meaning
LS RS	Loading of a single module and resolving a single function
LA RA	Loading of all the modules and resolving all of the functions
LA RS	Loading of all the modules but resolving only single function

Table 4.4: Modular application configurations. Source [37].

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

Table 4.5: Full execution time measurements. Source [37].

The results are shown in Table 4.5. In comparison to the monolithic approach, the LA RA and LA RS configurations are slightly slower and consume more resources. This is expected because module loading and their preparation for execution require additional steps, which involve additional logic introduced by our implementation. As anticipated, loading all the functionality and resolving all the functions is the most resource- and time-consuming task. Nevertheless, since we are interested in creating adaptive applications, which load functionality on demand, it might be unrealistic to load everything all at once. As a result, loading a single module and resolving only the needed functions shows that we can reduce the execution time and memory consumption significantly because only the required mechanisms are used.

4.4 System Overhead

As shown in the entire execution results, the implementation introduces additional latency. Similar to the previous measurements, the overhead of function calls is subjective as it depends on the contents of a function. However, to make functions comparable we employ small modules which contain a single function, which, upon its invocation returns a single integer. The results of the system overhead measurements are shown in Table 4.6. To measure the overhead we firstly investigate the overhead of host functions, *dlopen* and *dlsym*. For *dlsym*, we measure the time required to resolve a function, which has not been resolved previously, marked as *dlsym*. The *symbol* represents a function call, which has been resolved using *dlsym*. We also consider the average execution time of an imported function and how much overhead it introduces (*import*). Finally, we measure how long it takes for a module to execute its native function, marked in the table as *function*.

The overhead caused by the *dlopen* is the most significant. This can be limiting in the case of loading a relatively large module as the execution cannot proceed before the module is ready to be executed. The resolving of functions achieved with *dlsym* also takes a considerable amount of time. However, it is more efficient than *dlopen* due to the nature of mechanisms involved to resolve a function. For the function calls performing the actual processing, the execution of resolved function, *symbol*, takes longer to execute when compared to the *import* and *function* cases. This is due to the degree of indirection required to access the actual function. The last two remaining cases are very similar in terms of performance, however, the *import* does indeed introduce some additional latency.

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

Table 4.6: Function execution time measurements. Source [37].

4.5 Analysis of Results

With dynamic linking, we were able to divide monolithic applications into multiple modules. In the presented results, we decreased the startup time by almost 98% by decoupling the initial logic required to start the application. This also impacts the amount of memory consumed by the process. Moreover, by loading only the needed parts for the execution we have shown that execution time can be decreased by 89%.

As anticipated, module integration using the *dlopen* function is the most severe in terms of performance because of the additional operations it has to perform. Note that the time demanded by the *dlopen* function call will also depend heavily on the binary size. The larger the binary, the longer it will take for the module to be read, compiled, and instantiated. The *dlsym* function is also not of a static nature. It depends on the number of modules already loaded because *dlsym* walks through all the instantiated modules until it finds the needed function. Nonetheless, if a function has already been resolved by another module, *dlsym* will take significantly less time to resolve it again as a global table will already possess the index of a function. The capability to invoke functionality shared among multiple modules indirectly increases the time required to execute a function. In comparison to the direct *function* call, executing a function via a global table is almost three times slower.

The modularity of Webassembly applications can improve the way developers build applications targeting heterogeneous environments. Modules can be retrieved on-demand and integrated during the execution. As presented earlier in the chapter, the startup time can be reduced significantly and the redundant code can be decreased to the required amount at a particular time of execution. Such a technique can minimize excessive resource usage and improve user experience. Nonetheless, the implementation could still be improved in terms of performance. In the case of loading all the modules, and using all the available functionality, the runtime introduces additional overhead. This is currently unavoidable because of the implementation limitations, which we discuss in the following chapter.

5 Discussion

In this chapter, we revisit and provide answers to the research questions that were introduced at the beginning of the thesis. In particular, Section 5.1 provides answers to the first two research questions. The following, Section 5.2 and Section 5.3 addresses the third research question separating the limitations into two categories: the used technology – WebAssembly introduced in our previous work [37], and the implementation itself.

5.1 Research Questions Revisited

RQ1: What mechanisms enable adapting on-demand during execution using WebAssembly? During the implementation, we set out to achieve the adaptation to a particular environment using WebAssembly modules as the building blocks of the application. However, to achieve adaptation, the first goal was to extend WebAssembly capabilities as not all the needed features were available. As a result, the first mechanism can be identified as dynamic linking, which was introduced in one of our published papers [37]. Not only does dynamic linking enable the system to integrate modules on-demand, but the shared-everything approach creates the possibility for modules to share data and functionality, which otherwise would be isolated for each module.

Metadata is essential to retrieve the correct modules. Metadata acts as a scenario for the whole application – it defines the attributes of the system and modules that characterize the whole application. Furthermore, the module management library is responsible for communicating with remote repositories which contain the necessary functionality that the application requires. Not directly related to WebAssembly, however just as important as the previously mentioned mechanisms is the repository. Repository, in this case, serves as more than just simple storage for modules. Similar to the role of repository introduced in Section 2.2, the repository is responsible for module selection based on the available attributes. This alleviates the PoC implementation from unnecessary processing in case the application is running on resource-constrained devices and leaves the selection up to the remote entity. This is also more flexible as each repository can employ its selection algorithms.

RQ2: How do modular WebAssembly applications compare to monolithic WebAssembly applications? Modularization of monolithic WebAssembly applications increases the flexibility of software composition significantly. In this particular work, we can see that dividing the logic of the application empowers users to make decisions on-demand and integrate modules during execution. Besides that, it enables to use of only the necessary functionality instead of loading all the code with the risk of it never being used and redundantly using the available resources. The results provided in Chapter 4 show that the initial startup time is improved drastically. Moreover, the memory consumption has decreased substantially as the application starts as a relatively small module. However, the used approach does indeed introduce some additional overhead due to the module integration process which involves numerous actions, such as memory management, module preparation for execution, and performing symbol relocation.

5.2 Limitations of the Implementation

The implementation is by no means production-ready and is only a proof-of-concept. Some issues arise from the limitations of WebAssembly, while others are faults of the design decisions made throughout the implementation phase. As identified in the evaluation, the current API introduces a slight overhead. This, however, could be alleviated by a mechanism capable of loading modules before their actual execution.

The current memory model is also impacted by the capabilities of WebAssembly. Due to the absence of memory separation, we rely on a shared-everything memory model. In other words, we sacrifice security for flexibility. As described in Chapter 3, a module can access any information stored in the global memory because it comprises other module memory. Nevertheless, whenever the multi-memory proposal [56] becomes available, modules will be capable of referring to more than one memory component. With additional effort, such a mechanism could be extended to create protected memory areas and thus isolate sensitive data.

At the current state of the implementation, the information carried by the metadata is ultimately defined by the developer. However, it is a delicate component a majority of the system depends on. A minor change to the metadata could result in an unexpected outcome if a developer is not familiar with how the underlying mechanisms work. To elaborate, changing the interface ID of a described dependency to which a module

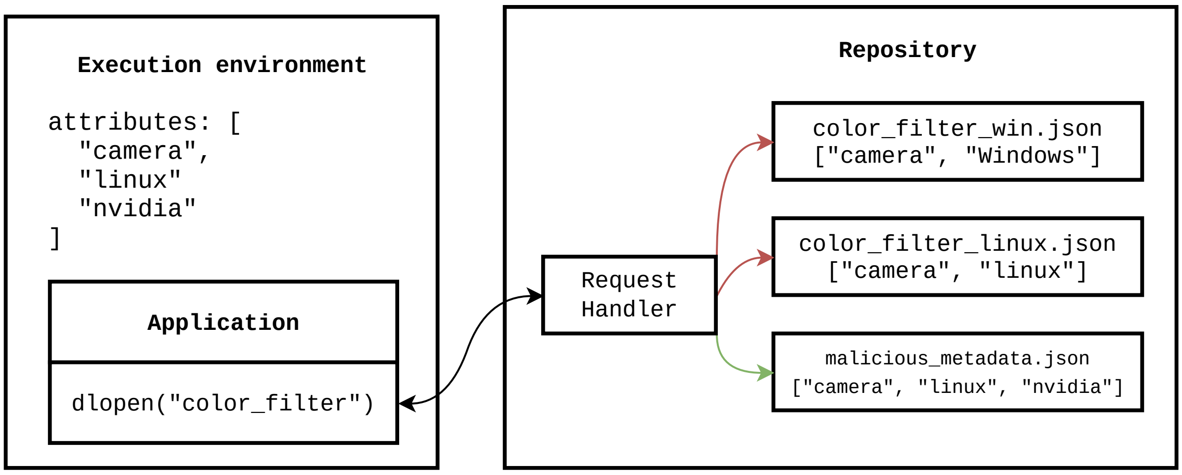
Execution environment
attributes: ["camera", "linux" "nvidia"]
Application
dlopen("color_filter")
Repository
Request Handler
color_filter_win.json ["camera", "Windows"]
color_filter_linux.json ["camera", "linux"]
malicious_metadata.json ["camera", "linux", "nvidia"]

Figure 5.1: Metadata hijacking.

should conform would impact the module selection process by instructing the repository to search for a completely different module.

Furthermore, in the case of attribute modification, the selection process might be misled and result in an inaccurate decision. A developer should not be permitted to add new or modify existing attributes nor should these be listed in the metadata, but rather generated automatically. This restriction could prevent compromising the module selection mechanism by supplying incorrect attributes and hijacking the application. To specify, without the restriction to modify the attribute set, an adversary could include attributes to prioritize the selection of that particular metadata which would lead to the retrieval of potentially malicious modules. An example case is shown in Figure 5.1 where an adversary plants a malicious metadata file in a remote repository. The malicious metadata file conforms to the same *color_filter* interface ID as the remaining two metadata files. During the selection, the repository evaluates the supported attributes of each aggregated metadata file and selects one satisfying the largest number of attributes. In this particular case, an adversary inserted an additional, unrelated *nvidia* attribute, which forces the selection mechanism to return the *malicious_metadata.json* as the best fit.

Another issue is related to dependencies. Currently, It is impossible to refer to a specific version of a module. This is further complicated by the fact that without versioning it cannot be distinguished whether any modifications were introduced to the module. Ultimately, this means that a user cannot be guaranteed what code the running application will receive from the repository. The absence of a versioning scheme enables modules to threaten the application because a module might break the program completely if it introduced breaking API changes in a newer version unknown to the user.

A more general issue regarding the metadata is its separation from a module. To retrieve a module, the current implementation performs two requests to either the same or different repositories. This results in additional overhead by issuing two requests to the repository instead of one. However, the issue could be avoided by taking advantage of the WebAssembly format.

As introduced in Chapter 2, WebAssembly employs custom sections to embed custom data. Moreover, differently than with the remaining sections, custom sections are not validated by the technology. As a result, metadata information could be embedded into the WebAssembly binary and analyzed in parallel after the retrieval. Such an approach could reduce the need to have metadata stored separately. Regarding the performance of the binary analysis, the sections can be iterated through quickly as the size component of each section indicates what is its size in bytes. Nonetheless, this enables filtering out only the custom sections. To distinguish whether a custom section holds metadata information, it can be given a name (e.g., *metadata*), which follows after the section indication byte. This approach enables to leap through the sections without the need to analyze their contents.

5.3 Limitations of WebAssembly

WebAssembly has already gained a lot of traction in research. Apart from being considered for different computing paradigms, research analyzing various properties of the technology had emerged. For example, Disselkoen et al. [14] noted that WebAssembly safety ensured by the sandboxed environment does not address potential memory vulnerabilities (e.g., buffer overflow) possible to be exploited by leveraging the current memory model. This is especially relevant to our approach, which embraces code reuse by blindly trusting modules from third-party vendors. Similarly, Lehmann et al. [34] list an extensive list of memory-related vulnerabilities that could be exploited, which include three dimensions of different attack types. Another issue is the lacking possibility to distinguish between public and private data due to the memory being a contiguous block of bytes. Hence, the data stored in an exported memory is available in its entirety to another module if imported. Based on these insights we consider the memory model of WebAssembly to be a significant limitation affecting the application quality.

Another significant limitation is performance. An execution environment is expected to possess certain characteristics to efficiently run WebAssembly applications [57]. Depending on the environment, it might not always be the case and result in additional overhead. Similarly, as described by Watt [55], the technology is currently only capable of single-threaded computing, which disables pushing the performance boundaries even further than initially anticipated.

As a part of the technology, we also consider VMs enabling to execute WebAssembly applications. Usually, the available VMs differ in their offered features and capabilities. Some are more suited for resource-constrained devices, where resource consumption matters, while others rely on providing excessive features, that might not work in a plethora of execution environments. Such VMs use contrasting APIs, and thus an application cannot use different VMs in a portable manner and requires to be re-engineered. Moreover, at least with the used VM – Wasmtime, releasing loaded modules is impossible at the current state of the VM. This is alarming because we no longer employ only what is necessary for the application and redundantly use the available resources. The last issue noticed about the used VM during the development regards module compilation. To reiterate, before proceeding with actual execution, downloaded modules are firstly compiled and instantiated. With the current JIT compilation, the performance is not as efficient as it could be with streaming compilation. WebAssembly binaries are represented by a format, which can be compiled as soon as the first blocks of data are received over the network [10]. Such a technique could improve the performance of the current solution significantly as it already proved to be useful in the context of the browser. Results published on Google Developers website [24] show that streaming compilers can reduce the overhead of compilation, which is performed after downloading a module.

6 Conclusions

We investigated WebAssembly technology as a potential candidate for creating adaptive applications. The applications are capable of starting as minimal base modules and evolving into complete applications. This is achieved by retrieving additional modules conforming to the requirements of the execution environment whenever they are needed. To achieve retrieving and integrating modules on-demand we built a lightweight runtime capable of retrieving modules from online repositories and linking them during execution. The introduced runtime was influenced by adaptive systems, package managers, and WebAssembly.

Modular WebAssembly applications alleviate the burden of shipping large binaries which not only impact the time required to start the application but also do not scale well. The results show that splitting a monolithic application into multiple building blocks can reduce the initial size of a binary significantly. This directly correlates with the startup time and memory requirements of the application. Moreover, modular WebAssembly applications enforce to use of only the code which is necessary for the execution, as a result, redundant functionality is minimized in comparison to the monolithic applications. Nonetheless, the implementation does indeed introduce some additional overhead whenever loading new modules. This, however, could be mitigated by investigating how the system could predict when to load modules in advance to minimize the potential overhead.

The implementation can be improved in a variety of aspects. Firstly, the current module description is not reliable because it is composed manually by the developer. Similarly, there is no standardized list of attributes (as it is done in Portage) that could be leveraged. Moreover, the selection of modules could be improved. As discussed, it is also not difficult to surpass such a system and potentially employ malicious code. Regarding the technology, some aspects are not mature enough to allow robust implementations. For example, the unloading of a module is crucial to remove unused functionality and maintain a reasonable binary size. This is especially important for resource-constrained devices. Similarly, regarding the WebAssembly VMs, because the format is designed to be streamable, leveraging streaming compilers could impact the startup time of monolithic and modular WebAssembly applications significantly.

However, most of the available VMs do not leverage such capability.

Looking further into the future, WebAssembly has already gained a lot of interest as an alternative technology for different computing environments. With the implementation presented in the thesis and prospects of future work, WebAssembly could become a new standard technology for building adaptive applications that could be leveraged in a multitude of different environments.

References

[1] P. Abate, R. Di Cosmo, G. Gousios, and S. Zacchiroli. “Dependency Solving Is Still Hard, but We Are Getting Better at It”. In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2020, pp. 547–551. DOI: 10.1109/SANER48275.2020.9054837.

[2] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. “A modular package manager architecture”. In: *Information and Software Technology* 55.2 (2013). Special Section: Component-Based Software Engineering (CBSE), 2011, pp. 459–474. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2012.09.002. URL: https://www.sciencedirect.com/science/article/pii/S0950584912001851.

[3] P. Abate, R. DiCosmo, R. Treinen, and S. Zacchiroli. “MPM: A Modular Package Manager”. In: *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*. CBSE ’11. Boulder, Colorado, USA: Association for Computing Machinery, 2011, pp. 179–188. ISBN: 9781450307239. DOI: 10.1145/2000229.2000255. URL: https://doi-org.libproxy.helsinki.fi/10.1145/2000229.2000255.

[4] G. Amadio and B. Xu. “Portage: Bringing Hackers’ Wisdom to Science”. In: *CoRR* abs/1610.02742 (2016). arXiv: 1610.02742. URL: http://arxiv.org/abs/1610.02742.

[5] Appcypher. *Awesome WebAssembly Runtimes*. 2021. URL: https://github.com/appcypher/awesome-wasm-runtimes (visited on 11/21/2020).

[6] O. Bal-Pétré, P. Varlez, and F. Perez-Tellez. “Pacloud: Towards a Universal Cloud-Based Linux Package Manager”. In: *Proceedings of the 2019 International Communication Engineering and Cloud Computing Conference*. CECCC 2019. Prague, Czech Republic: Association for Computing Machinery, 2019, pp. 6–13. ISBN: 9781450376396. DOI: 10.1145/3380678.3380685. URL: https://doi-org.libproxy.helsinki.fi/10.1145/3380678.3380685.

[7] N. Belaramani, C.-L. Wang, and F. Lau. “Dynamic component composition for Functionality Adaptation in pervasive environments”. In: *The Ninth IEEE Work-*

shop on Future Trends of Distributed Computing Systems, 2003. FTDCS 2003. Proceedings. 2003, pp. 226–232. DOI: 10.1109/FTDCS.2003.1204338.

[8] F. Brown, A. Mirian, A. Jaiswal, A. Notzli, and D. Stefan. "SPAM: a Secure Package Manager". In: (2017).

[9] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman. "A Look in the Mirror: Attacks on Package Managers". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security.* CCS '08. Alexandria, Virginia, USA: Association for Computing Machinery, 2008, pp. 565–574. ISBN: 9781595938107. DOI: 10.1145/1455770.1455841. URL: https://doi-org.libproxy.helsinki.fi/10.1145/1455770.1455841.

[10] W. W. W. Consortium. *WebAssembly Core Specification.* 2019. URL: https://www.w3.org/TR/wasm-core-1/.

[11] U. Dastgeer and C. Kessler. "Conditional component composition for GPU-based systems". In: *Proc. Seventh Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-2014) at HiPEAC-2014, Vienna, Austria, Jan. 2014.* HiPEAC NoE, 2014. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-114340.

[12] L. Deri. "Droplets: Breaking Monolithic Applications Apart". In: (1995).

[13] J. Dieguez Castro. "Gentoo Linux". In: *Introducing Linux Distros.* Berkeley, CA: Apress, 2016, pp. 253–272. ISBN: 978-1-4842-1392-6. DOI: 10.1007/978-1-4842-1392-6_12. URL: https://doi.org/10.1007/978-1-4842-1392-6_12.

[14] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan. "Position Paper: Progressive Memory Safety for WebAssembly". In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy.* HASP '19. Phoenix, AZ, USA: Association for Computing Machinery, 2019. ISBN: 9781450372268. DOI: 10.1145/3337167.3337171. URL: https://doi-org.libproxy.helsinki.fi/10.1145/3337167.3337171.

[15] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering.* Ed. by M. Mazzara and B. Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. URL: https://doi.org/10.1007/978-3-319-67425-4_12.

[16] emscripten-core. *Linking*. 2021. URL: https://github.com/emscripten-core/emscripten/wiki/Linking (visited on 03/10/2020).

[17] F. Fjellheim. “Over-the-air deployment of applications in multi-platform environments”. In: *Australian Software Engineering Conference (ASWEC’06)*. 2006, 10 pp.–170. DOI: 10.1109/ASWEC.2006.39.

[18] M. Franz and T. Kistler. “Slim Binaries”. In: *Commun. ACM* 40.12 (Dec. 1997), pp. 87–94. ISSN: 0001-0782. DOI: 10.1145/265563.265576. URL: https://doi-org.libproxy.helsinki.fi/10.1145/265563.265576.

[19] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer. “Sledge: A Serverless-First, Light-Weight Wasm Runtime for the Edge”. In: *Proceedings of the 21st International Middleware Conference*. Middleware ’20. Delft, Netherlands: Association for Computing Machinery, 2020, pp. 265–279. ISBN: 9781450381536. DOI: 10.1145/3423211.3425680. URL: https://doi-org.libproxy.helsinki.fi/10.1145/3423211.3425680.

[20] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. “The Spack package manager: bringing order to HPC software chaos”. In: *SC ’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12. DOI: 10.1145/2807591.2807623.

[21] C. J. M. Geisterfer and S. Ghosh. “Software component specification: a study in perspective of component selection and reuse”. In: *Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS’05)*. 2006, p. 9. DOI: 10.1109/ICCBSS.2006.26.

[22] Gentoo Linux. *USE flag index*. 2021. URL: https://www.gentoo.org/support/use-flags/ (visited on 04/02/2021).

[23] Gentoo Linux Wiki. *Portage*. 2021. URL: https://wiki.gentoo.org/wiki/Portage (visited on 04/02/2021).

[24] Google Developers. *Loading WebAssembly modules efficiently*. 2018. URL: https://developers.google.com/web/updates/2018/04/loading-wasm (visited on 05/19/2020).

[25] R. Gurdeep Singh and C. Scholliers. “WARDuino: A Dynamic WebAssembly Virtual Machine for Programming Microcontrollers”. In: MPLR 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 27–36. ISBN: 9781450369770. DOI: 10.1145/3357390.3361029. URL: https://doi-org.libproxy.helsinki.fi/10.1145/3357390.3361029.

[26] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. “Bringing the Web up to Speed with WebAssembly”. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 185–200. ISSN: 0362-1340. DOI: 10.1145/3140587.3062363. URL: https://doi-org.libproxy.helsinki.fi/10.1145/3140587.3062363.

[27] A. Hall and U. Ramachandran. “An Execution Model for Serverless Functions at the Edge”. In: *Proceedings of the International Conference on Internet of Things Design and Implementation.* IoTDI ’19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 225–236. ISBN: 9781450362832. DOI: 10.1145/3302505.3310084. URL: https://doi.org/10.1145/3302505.3310084.

[28] A. R. Hevner, S. T. March, J. Park, and S. Ram. “Design Science in Information Systems Research”. In: *MIS Q.* 28.1 (Mar. 2004), pp. 75–105. ISSN: 0276-7783.

[29] M. Jacobsson and J. Willén. “Virtual Machine Execution for Wearables Based on WebAssembly”. In: *13th EAI International Conference on Body Area Networks.* Ed. by C. Sugimoto, H. Farhadi, and M. Hämäläinen. Cham: Springer International Publishing, 2020, pp. 381–389. ISBN: 978-3-030-29897-5.

[30] M. Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook.* 1st. USA: No Starch Press, 2010. ISBN: 1593272200.

[31] M. Kicherer. “Reducing the Complexity of Heterogeneous Computing: A Unified Approach for Application Development and Runtime Optimization”. In: (2014).

[32] M. Kicherer, F. Nowak, R. Buchty, and W. Karl. “Seamlessly Portable Applications: Managing the Diversity of Modern Heterogeneous Systems”. In: *ACM Trans. Archit. Code Optim.* 8.4 (Jan. 2012). ISSN: 1544-3566. DOI: 10.1145/2086696.2086721. URL: https://doi.org/10.1145/2086696.2086721.

[33] X. Larrucea, A. Combelles, J. Favaro, and K. Taneja. “Software Engineering for the Internet of Things”. In: *IEEE Software* 34.1 (2017), pp. 24–28. DOI: 10.1109/MS.2017.28.

[34] D. Lehmann, J. Kinder, and M. Pradel. “Everything Old is New Again: Binary Security of WebAssembly”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 217–234. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

[35] LLVM Documentation. *WebAssembly lld port*. 2021. URL: https://lld.llvm.org/WebAssembly.html (visited on 05/22/2021).

[36] J. Long, H. .-Y. Tai, S. .-T. Hsieh, and M. J. Yuan. “A Lightweight Design for Serverless Function as a Service”. In: *IEEE Software* 38.1 (2021), pp. 75–80. DOI: 10.1109/MS.2020.3028991.

[37] N. Mäkitalo, V. Bankowski, P. Daubaris, R. Mikkola, O. Beletski, and T. Mikkonen. “Bringing WebAssembly up to Speed with Dynamic Linking”. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. SAC ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 1727–1735. ISBN: 9781450381048. DOI: 10.1145/3412841.3442045. URL: https://doi.org/10.1145/3412841.3442045.

[38] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. “Composing adaptive software”. In: *Computer* 37.7 (2004), pp. 56–64. DOI: 10.1109/MC.2004.48.

[39] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng. “A taxonomy of compositional adaptation”. In: *Rapport Technique numéroMSU-CSE-04-17* (2004).

[40] P. Mendki. “Evaluating Webassembly Enabled Serverless Approach for Edge Computing”. In: *2020 IEEE Cloud Summit*. 2020, pp. 161–166. DOI: 10.1109/IEEECloudSummit48914.2020.00031.

[41] A. Miranda and J. Pimentel. “On the Use of Package Managers by the C++ Open-Source Community”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC ’18. Pau, France: Association for Computing Machinery, 2018, pp. 1483–1491. ISBN: 9781450351911. DOI: 10.1145/3167132.3167290. URL: https://doi-org.libproxy.helsinki.fi/10.1145/3167132.3167290.

[42] Node Package Manager. 2021. URL: https://www.npmjs.com (visited on 02/04/2021).

[43] npm. *Package Metadata.* 2021. URL: https://github.com/npm/registry/blob/master/docs/responses/package-metadata.md (visited on 10/14/2020).

[44] C. Raibulet. “Facets of Adaptivity”. In: *Software Architecture.* Ed. by R. Morrison, D. Balasubramaniam, and K. Falkner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 342–345. ISBN: 978-3-540-88030-1.

[45] R. E. V. Rosa and V. F. Lucena. “Smart Composition of Reusable Software Components in Mobile Application Product Lines”. In: *Proceedings of the 2nd International Workshop on Product Line Approaches in Software Engineering.* PLEASE ’11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 45–49. ISBN: 9781450305846. DOI: 10.1145/1985484.1985496. URL: https://doi.org/10.1145/1985484.1985496.

[46] S. S. Salim, A. Nisbet, and M. Luján. “TruffleWasm: A WebAssembly Interpreter on GraalVM”. In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments.* VEE ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 88–100. ISBN: 9781450375542. DOI: 10.1145/3381052.3381325. URL: https://doi-org.libproxy.helsinki.fi/10.1145/3381052.3381325.

[47] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. “Software diversity: State of the art and perspectives”. In: *International Journal on Software Tools for Technology Transfer* 14 (Oct. 2012). DOI: 10.1007/s10009-012-0253-y.

[48] S. Shillaker and P. Pietzuch. “Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing”. In: *2020 USENIX Annual Technical Conference (USENIX ATC 20).* USENIX Association, July 2020, pp. 419–433. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shillaker.

[49] D. Spinellis. “Package Management Systems”. In: *IEEE Software* 29.2 (2012), pp. 84–86. DOI: 10.1109/MS.2012.38.

[50] C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming.* Pearson Education, 2002.

[51] A. Taivalsaari and T. Mikkonen. “A Taxonomy of IoT Client Architectures”. In: *IEEE Software* 35.3 (2018), pp. 83–88. DOI: 10.1109/MS.2018.2141019.

[52] V8. *Liftoff: a new baseline compiler for WebAssembly in V8*. 2018. URL: https://v8.dev/blog/liftoff (visited on 04/13/2021).

[53] Wasmtime. *Struct wasmtime::Linker*. 2021. URL: https://docs.rs/wasmtime/0.21.0/wasmtime/struct.Linker.html (visited on 12/10/2020).

[54] C. Watt. "Mechanising and Verifying the WebAssembly Specification". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Los Angeles, CA, USA: Association for Computing Machinery, 2018, pp. 53–65. ISBN: 9781450355865. DOI: 10.1145/3167082. URL: https://doi.org/10.1145/3167082.

[55] C. Watt, A. Rossberg, and J. Pichon-Pharabod. "Weakening WebAssembly". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360559. URL: https://doi.org/10.1145/3360559.

[56] WebAssembly. *Multi Memory Proposal for WebAssembly*. 2021. URL: https://github.com/WebAssembly/multi-memory (visited on 11/21/2020).

[57] WebAssembly. *Portability: Assumptions for Efficient Execution*. 2021. URL: https://webassembly.org/docs/portability/#assumptions-for-efficient-execution (visited on 03/29/2021).

[58] WebAssembly. *WebAssembly Dynamic Linking*. 2021. URL: https://github.com/WebAssembly/tool-conventions/blob/master/DynamicLinking.md (visited on 05/11/2020).

[59] E. Wen and G. Weber. "Wasmachine: Bring IoT up to Speed with A WebAssembly OS". In: *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2020, pp. 1–4. DOI: 10.1109/PerComWorkshops48775.2020.9156135.

[60] E. Wittern, P. Suter, and S. Rajagopalan. "A Look at the Dynamics of the JavaScript Package Ecosystem". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 351–361. ISBN: 9781450341868. DOI: 10.1145/2901739.2901743. URL: https://doi-org.libproxy.helsinki.fi/10.1145/2901739.2901743.

[61] K. Zandberg and E. Baccelli. “Minimal Virtual Machines on IoT Microcontrollers: The Case of Berkeley Packet Filters with rBPF”. In: *2020 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN)*. IEEE. 2020, pp. 1–6.