Browserify: Empowering Consistent and Efficient Application Deployment across Heterogeneous Mobile Devices

Elliott Wen

A thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Science,
The University of Auckland, 2020.

Abstract

Nowadays, mobile devices have become an integral part of daily life. Traditionally, mobile applications are developed natively using platform-specific programming languages and development environments. To enable a native application to run on multiple platforms is not a simple task. Usually, this would require dedicated streams of development for each platform. With the emergence of more and more mobile platforms, the effort required to provide multi-platform support becomes strenuous. To address this problem, software engineering communities propose a cross-platform development approach. Cross-platform development enables the use of a single codebase across multiple platforms while providing acceptable application performance. This is mainly achieved using web technologies (e.g., HTML, CSS, and JavaScript) that are inherently portable from one mobile platform to another.

However, existing cross-platform development frameworks are still struggling with several challenges. Firstly, they lack interoperability with commonly-used programming languages. For a long time, JavaScript was the only supported programming language by these frameworks. It is generally challenging to reuse a wide range of existing programs or libraries written in other programming languages. Secondly, most cross-platform apps run significantly slower than native applications. A primary culprit is the use of Javascript, which is notoriously hard to compile and optimize efficiently. The performance penalty degrades user experience and can render applications unusable on low-end devices. The recent rise of the WebAssembly has shed some light on the two challenges. WebAssembly is a young low-level programming language supported by every modern browser. WebAssembly promises to deliver significantly higher performance than Javascript and is designed to have high interoperability with

many popular programming languages. Nevertheless, WebAssembly itself does not provide a complete solution to the challenges above. Compared to native applications, WebAssembly applications have been reported on average to run 45% slower according to previous benchmarks. Besides that, many existing programs can not be directly reused since they expect abstractions from modern OSs such as threads, file system, and sockets. It is important to note that these abstractions are not directly available in WebAssembly.

In this thesis, we present Browserify, a framework that enables consistent and performant application deployment across heterogeneous mobile devices. Browserify consists of two major components: BroswerVM and Wasmachine. BrowserVM is a WebAssembly-based virtual machine hypervisor that allows the execution of unmodified and complete operating systems and applications inside browsers. Though slower than native hypervisors, BroswerVM provides acceptable performance to reuse unmodified applications that are not compute-intensive. Wasmachine is an OS aiming to efficiently and securely execute WebAssembly applications in mobile devices with constrained resources. Wasmachine achieves efficient execution by compiling WebAssembly ahead of time to native binary and executing it in kernel mode for zero-cost system calls. In practice, WebAssembly applications in Wasmachine can even run faster than their native counterparts in Linux. We illustrate the capabilities of Browserify via a case study called SwiftPad. In SwiftPad, we port the famous TEX typesetting system to e-paper mobile devices and enable users to compose high typographic quality documents in a WYSIWYG (what you see is what you get) fashion. We showcase how Browserify facilitates building cross-platform applications and the reuse of existing software components.

Acknowledgment

Throughout the writing of this thesis, I have received a great amount of support and assistance. I would first like to thank my supervisors, Dr Gerald Weber and Professor Jim Warren, who made this work possible. Their friendly and insightful guidance has been invaluable throughout all stages of the work. I also want to express my gratitude to Kylie, who provides a professional and free proofreading service. Special thank goes to my parents, who have been patient over the years and never raised an eyebrow whenever I claimed my thesis would be finished 'in the next month' for nearly a year.

School of Graduate Studies
AskAuckland Central
Alfred Nathan House
The University of Auckland
Tel: +64 9 373 7599 ext 81321
Email: postgradinfo@auckland.ac.nz

Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work**. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work. SwiftLaTeX: Exploring Web-based True WYSIWYG Editing for Digital Publishing in Proceedings of the ACM Symposium on Document Engineering 2018	
Nature of contribution by PhD candidate	System Implementation and Publication
Extent of contribution by PhD candidate (%)	75

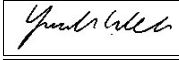
CO-AUTHORS

Name	Nature of Contribution
Gerald Weber	Supervision and Paper Revision

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

Name	Signature	Date
Gerald Weber		2020-10-27

Last updated: 28 November 2017

School of Graduate Studies
AskAuckland Central
Alfred Nathan House
The University of Auckland
Tel: +64 9 373 7599 ext 81321
Email: postgradinfo@auckland.ac.nz

Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work**. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work. PaperWork: Exploring the Potential of Electronic Paper on Office Work in Proceedings of the ACM Symposium on Document Engineering 2019	
Nature of contribution by PhD candidate	System Implementation and Publication
Extent of contribution by PhD candidate (%)	80

CO-AUTHORS

Name	Nature of Contribution
Gerald Weber	Supervision and Paper Revision
Jim Warren	Supervision and Paper Revision

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

Name	Signature	Date
Gerald Weber		2020-10-27
Jim Warren		2020-10-29

Last updated: 28 November 2017

School of Graduate Studies
AskAuckland Central
Alfred Nathan House
The University of Auckland
Tel: +64 9 373 7599 ext 81321
Email: postgradinfo@auckland.ac.nz

Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work**. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work. BroswerVM: Running unmodified operating systems and applications in browsers in IEEE International Conference on Web Services (IEEE ICWS 2020)	
Nature of contribution by PhD candidate	System Implementation and Publication
Extent of contribution by PhD candidate (%)	80

CO-AUTHORS

Name	Nature of Contribution
Gerald Weber	Supervision and Paper Revision
Jim Warren	Supervision and Paper Revision

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

Name	Signature	Date
Gerald Weber		2020-10-27
Jim Warren		2020-10-29

Last updated: 28 November 2017

School of Graduate Studies
AskAuckland Central
Alfred Nathan House
The University of Auckland
Tel: +64 9 373 7599 ext 81321
Email: postgradinfo@auckland.ac.nz

Co-Authorship Form

This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work**. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements. Co-authored works may be included in a thesis if the candidate has written all or the majority of the text and had their contribution confirmed by all co-authors as not less than 65%.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work. Elliott Wen, Gerald Weber. "Wasmachine: Bring IoT up to Speed with A WebAssembly OS". In 2020 IEEE International Conference on Cloud Computing (IEEE Cloud 2020).

Nature of contribution by PhD candidate	System Implementation and Publication
Extent of contribution by PhD candidate (%)	80

CO-AUTHORS

Name	Nature of Contribution
Gerald Weber	Supervision and Paper Revision

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- that the candidate wrote all or the majority of the text.

Name	Signature	Date
Gerald Weber		2020-10-27

Contents

Contents **11**

List of Figures **13**

List of Tables **15**

1 Introduction **17**
- 1.1 Native Development and Device Fragmentation 17
- 1.2 Embracing Cross-platform Development 18
- 1.3 Web-based Cross-platform Development 19
- 1.4 WebAssembly-based Cross-platform Development 21
- 1.5 Main Contributions 22
- 1.6 Publications arising from this thesis 23
- 1.7 Organization of thesis 24

2 Literature Review **27**
- 2.1 Ubiquitous Mobile Devices 27
- 2.2 Native Development 28
- 2.3 Cross-platform Development 29

3 BrowserVM: Running Unmodified Operating Systems and Applications in Browsers **41**
- 3.1 Rationale 42
- 3.2 Background 44
- 3.3 Binary Translation to WebAssembly 47
- 3.4 Optimizing WebAssembly Execution 51

3.5 Cache sharing among different runs 54
3.6 Hardware Emulation 55
3.7 System Evaluation 62
3.8 Chapter Summary 65

4 Wasmachine: Bring the Edge up to Speed with A WebAssembly OS **67**
4.1 Motivation 68
4.2 Background 71
4.3 AOT Compilation for WebAssembly 75
4.4 WebAssembly-optimized Kernel Implementation 79
4.5 System Evaluation 82
4.6 Chapter Summary 90

5 Case Study: Exploring WYSIWYG LaTeX Editing on E-Paper **91**
5.1 Motivation 91
5.2 System Overview 94
5.3 Comparison with existing systems 96
5.4 Building Cross-platform SwiftPad 99
5.5 Making LaTeX WYSIWYG 102
5.6 Making SwiftPad Responsiveness 114
5.7 Optimizing the E-paper Display Driver 117
5.8 Chapter Summary 121

6 Conclusion and Discussion **123**
6.1 Future Research on BrowserVM 123
6.2 Future Research on Wasmachine 126
6.3 Future Directions on SwiftPad 128

Bibliography **1**

List of Figures

3.1 An example of unstructured control flow, where basic blocks are connected with branching edges. Numbers reflect the topological order of each block. 47
3.2 The work flow of BrowserVM's generational block management scheme. 53
3.3 BrowserVM enables guest applications to use WebGL to accelerate 3D rendering. 59
3.4 Relative Execution Time of Seven SPEC CPU Benchmarks 61
3.5 Relative execution time for different optimization techniques. . . . 65

4.1 Architecture of Conventional WebAssembly Runtimes 69
4.2 Offline Compilation and Architecture of Wasmachine 70
4.3 JIT V8 vs. AOT Wasmachine vs. Native Performance Comparison . 83
4.4 Performance Comparison of Kernel-intensive applications 85
4.5 Kernel time ratios in different runtimes 88
4.6 Performance Comparison among Rust-based Kernel and C-based Kernels 90

5.1 General Hardware Setup 94
5.2 User Interface of SwiftPad. 95
5.3 Project Management Interface of SwiftPad. 97
5.4 Three Editing States in the WYSIWYG View. 104
5.5 Timing Charts in the WYSIWYG View. 105
5.6 Workflow for the WYSIWYG view. 106
5.7 Two typical examples for source code position inference. 107
5.8 Internal Mechanism of the Engine Patch. 109
5.9 PDF to HTML conversion 114

5.10 Multi-Queue scheduling algorithm 119
5.11 How our algorithm takes advantage of the simultaneous update feature of the EPDC. 120

List of Tables

3.1 X86 arithmetic instructions V.S. WebAssembly arithmetic instructions 49

4.1 AriteConversion from WebAssembly to LLVM Bytecode 77
4.2 Test cases for AOT vs. JIT compilation 83

Chapter 1

Introduction

Recent decades have witnessed an explosive growth of mobile devices such as smartphones, wearables, and tablets. It is predicted that the total number of mobile devices will reach 5.6 billion, and approximately 90% of adults in developed countries will own at least one mobile device by 2023 [94]. Nowadays, mobile devices are commonly equipped with high computing power, large storage capacities, and wireless network access. These features as well as the device's innate portability and low cost have turned mobile devices into a dominant computing platform for many people.

1.1 Native Development and Device Fragmentation

Traditionally, mobile app development operates on a per-platform basis; apps are written using tools and languages specifically designed for each platform. For example, developing native apps for the Android platform requires knowledge of Android Studio development environment and the Java programming language. In contrast, developing apps for the iOS platform requires proficiency in the Xcode development environment and the Objective-C programming language [141]. This type of development is commonly referred to as the *Native* development approach [68]. Native apps developed for a mobile platform cannot be deployed to or be executed on other platforms. An Android app cannot run

on iOS and vice versa. This incompatibility is mainly attributed to inherent differences in hardware architectures, compiling techniques, and Application Programming Interfaces (API). Consequently, if a native app wishes to reach audiences across multiple platforms, it has to be written multiple times despite having the same core functionalities. This process is notoriously labor-intensive, partially due to the increasing fragmentation of mobile platforms (e.g., Android, iOS, Windows 10 Mobile, and Fire OS). Another issue with this approach is that it results in multiple inherently separate codebases. Maintaining multiple separate codebases with similar functionality can be a very tedious and error-prone task [53]. The practical implications are that there can be large overheads for updates, patches and bug fixes as changes need to be replicated to all platform-specific codebases.

1.2 Embracing Cross-platform Development

As the limitations of the native development approach come to light, many researchers are actively exploring an alternative development approach, which is commonly referred to as *cross-platform* development. The main strength of cross-platform development is 'write once and run everywhere'. This allows a single codebase to be reused across multiple platforms while providing acceptable application performance [141].

An early technique for cross-platform development is Syntax Directed Source-to-source Translation. One example is Google's J2ObjC [11]. It converts Java source code for Android into Objective-C source code for iOS. However, this approach is only applicable if the source and target language share similar syntax and programming paradigms. This approach also fails to convert platform-specific APIs. Some previous research works (e.g., Native-2-native [25] and Java-to-Swift [7]) attempt platform-specific APIs conversion using recommendation systems. They identify the semantic content contained in a block of source code, then formulate a query to search for the equivalent target code block using web-based programming resources such as StackOvewFlow and Github Issues [130]. Nevertheless, the generated conversion results are still prone to errors and require manual verification from developers.

Another technique for cross-platform development is Model-driven development (MDD). Here an application's functionality is represented by high-level, platform-independent declarative languages. These languages are commonly referred to as Domain-Specific Languages (DSL). To implement apps across mobile platforms, developers are only required to be proficient in one DSL language. Unfortunately, most DSL languages are still in their experimental stage and often lack clear documentation. This inevitably makes it hard for developers to learn [75]. Most MDD frameworks invent their own distinct languages. This means that the knowledge transferability between MDD frameworks is very low or non-existent. Another main drawback of MDD frameworks is their low language interoperability; they usually only allow developers to express application logic in DSLs. It is generally challenging to incorporate existing software components written in other programming languages. As a consequence, developers may have to implement many commonly-used functionalities from scratch.

In reality, cross-platform development only began to gain traction when a more promising web-based approach emerged.

1.3 Web-based Cross-platform Development

The web-based approach enables developers to implement cross-platform apps using web programming languages such as HTML5, CSS3, and JavaScript. Web-based apps possess a wide range of benefits. Firstly, they are effortlessly portable from one mobile platform to another because nearly every mobile platform has an in-built browser. The browsers generally have embraced the HTML5 standard and guarantee consistent behavior regardless of the underlying platform. Secondly, mobile browsers provide the capability to access rich device features such as geo-localization, cameras, storage, and multimedia. These features enable developers to implement more complex functionalities similar to native apps. Thirdly, unlike DSLs, web programming languages possess a more mature and increasingly growing community. Web developers can have easy access to documents and technical support from the Internet.

Although the web-based approach has significantly improved the adoption of cross-platform development, two key challenges are yet to be solved.

Interoperability. For a long time, JavaScript has been the only programming language in Web-based frameworks. This renders incorporating software components written in other programming languages tricky. The motivation for running other programming languages can be multi-fold. For example, a software company may have a large amount of existing code already written in another language, or a developer may have a strong preference and hence be more productive in another language. As a workaround, there are tools that compile certain languages to Javascript ([35] and [82]). However, these tools usually only allow a subset of the original language to be converted [143]. For example, it is notoriously hard to convert multi-threaded code to Javascript because Javascript is asynchronous and does not have a direct equivalence to threads. Some research [50] even suggest against treating Javascript as a compilation target due to its semantic pitfalls and inconsistent performance.

How to effectively integrate and reuse general programming languages in web-based cross-platform development remains challenging.

Software Performance. The performance of web-based apps is a topic frequently investigated in both academia and the industry. A majority of research [6, 18] suggest that web-based frameworks commonly lead to performance degradation in mobile apps. The performance penalty is generally acceptable on high-end devices. However, this is not the case for low-end devices where the apps can be rendered unusable. The slowdown is attributed to the difficulty in compiling and optimizing JavaScript. Javascript is thus not suitable for building demanding applications such as interactive 3D visualization, video software, and games [12, 113, 50].

How to build performant web-based applications for resource-constrained mobile devices remains an open research question.

1.4 WebAssembly-based Cross-platform Development

The recent rise of WebAssembly has shed light on the above challenges. WebAssembly is a low-level bytecode programming language supported by most modern browsers [78]. From the beginning, WebAssembly was designed as a universal compiler target. So, it is unsurprising that most commonly used compiler toolchains (C++, Rust, Go) can also produce WebAssembly binaries. WebAssembly also has an ambitious design target to deliver an execution speed as fast as native apps. Although still in its infancy, WebAssembly is gathering momentum and will be a key computing platform for years to come.

On the surface, it may seem that WebAssembly is what we all need to solve the research challenges in web-based cross-platform development. However, there are still some missing pieces to the puzzle. Firstly, although compilers are capable of producing WebAssembly, the compilation process can encounter frequent failures in practice. The reason behind this is that the compiler designers overlook a critical fact: most popular programming languages and their standard libraries expect abstractions from modern OSs such as threads, filesystem, and sockets. These abstractions are not directly available and can be hard to emulate inside browsers. A major consequence is that existing codebases must undergo substantial changes to accommodate the missing abstraction problem. Moreover, this approach would not work if developers do not have access to source code.

Secondly, applications compiled to WebAssembly still run slower by an average of 45% as reported by previous benchmarks [64]. The slowdown is attributed to two factors. Firstly, conventional WebAssembly runtimes translate WebAssembly instructions to native machine code using just-in-time compilers. They do not apply complex code optimization and tend to generate poor-quality output. Secondly, system calls from WebAssembly applications have to be proxy-ed by runtimes to reach operating systems. This process triggers frequent system context switching and incurs significant performance overhead.

The academic body of knowledge on WebAssembly is still limited to a small number of published works. Therefore, we aim to carry more research on

technical innovation, conceptual discussions, and case studies of WebAssembly, especially within web-based cross-platform development.

1.5 Main Contributions

This thesis presents Browserify, a framework that enables consistent and performant application deployment across heterogeneous mobile devices. Browserify consists of two major components: BroswerVM and Wasmachine. They provide workable solutions to the interoperability and performance challenges in web-based cross-platform development.

BrowserVM is a WebAssembly-based virtual machine hypervisor, which allows the execution of unmodified and complete operating systems and applications inside browsers. BrowserVM efficiently conducts processor emulation through dynamic binary translation. It also provides performant hardware emulation for hard disks, graphic cards, and network adapters. Though slower than native hypervisors, BroswerVM provides acceptable performance to reuse existing software components that are not compute-intensive. Additionally, BrowserVM requires no access to the source code.

Wasmachine is an OS aiming to efficiently and securely execute WebAssembly applications in mobile devices with constrained resources. Wasmachine translates WebAssembly instructions to native machine code using an ahead-of-time compiler. The compiler can apply sophisticated code optimization and produce efficient binaries with software-based fault isolation. The resulting binaries can be directly executed as a kernel thread to harness zero-cost system calls. We conduct a quantitative performance evaluation on our Wasmachine prototype. The results show that WebAssembly applications running in Wasmachine can even run faster than their native Linux counterparts.

To demonstrate the capabilities of Browserify, we also provide a case study called SwiftPad. In the study, we introduce a document composition system named SwiftPad for e-paper. SwiftPad renovates the famous TEX typesetting system, enabling users to compose high typographic quality documents on e-paper in a WYSIWYG (what you see is what you get) fashion. Building

such a cross-platform system on resource-constrained e-paper entails unique challenges. The study showcases how Browserify facilitates our implementation.

1.6 Publications arising from this thesis

The following is a list of publications that arose from the research described in this thesis. The contents of these publications appear in several chapters throughout this thesis.

1. Wen, E., & Weber, G. (2018, August). SwiftLaTeX: Exploring Web-based True WYSIWYG Editing for Digital Publishing. In Proceedings of the ACM Symposium on Document Engineering 2018 (pp. 1-10).

2. Wen, Elliott, Jim Warren, and Gerald Weber. "PaperWork: Exploring the Potential of Electronic Paper on Office Work." Proceedings of the ACM Symposium on Document Engineering 2019. 2019.

3. Wen, E., & Weber, G. (2018, October). Going Grey: Exploring the Potential of Electrophoretic Displays. In Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers (pp. 1761-1764).

4. Wen, E., & Weber, G. (2020, March). Wasmachine: Bring IoT up to Speed with A WebAssembly OS. In 2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops) (pp. 1-4). IEEE.

5. Wen, E., & Weber, G. (2020, March). SwiftPad: Exploring WYSIWYG TEX Editing on Electronic Paper. In 2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops) (pp. 1-4). IEEE.

6. Wen, E., & Weber, G. (2020, October). Wasmachine: Bring the Edge up to Speed with A WebAssembly OS. In 2020 IEEE International Conference on Cloud Computing IEEE.

7. Wen, E., Warren, J., & Weber, G. (2020, October). BrowserVM: Running Unmodified Operating Systems and Applications in Browsers. In 2020 IEEE International Conference on Web Services IEEE.

1.7 Organization of thesis

We organize this thesis as follows.

Chapter 1: Introduction. The first chapter provides an overview of the thesis, including the work's motivation and the problems addressed in the thesis.

Chapter 2: Literature Review. This chapter presents a literature review on cross-platform development. It includes a categorization of popular cross-platform approaches and discusses their advantages and drawbacks.

Chapter 3: Running Unmodified Operating Systems and Applications in Browsers. In this chapter, we present BroswerVM, a new approach to run unmodified and complete operating systems and applications inside browsers. BrowserVM aims to address the Interoperability issue in web-based cross-platform development.

Chapter 4: Bring the Resource-Constrained Devices up to Speed with A WebAssembly OS. In this chapter, we demonstrate Wasmachine, an OS aiming to efficiently and securely execute WebAssembly applications in mobile platforms with constrained resources. This chapter provides a workable solution to the performance issue in web-based cross-platform development.

Chapter 5: A Case Study: Exploring Web-based True WYSIWYG Editing for Digital Publishing. This chapter presents a case study called SwiftPad, where we compile the famous TEX typesetting system to WebAssembly and enable WYSIWYG document editing on e-paper. This chapter showcases how Browserify eases porting legacy applications to a resource-constrained mobile platform and enables new functionalities.

Chapter 5: Discussion and Conclusions. This chapter summarizes the key findings and discusses the limitations of our research. It also suggests directions for future work.

Chapter 2

Literature Review

This chapter provides a literature review of mobile application development approaches and discusses their advantages and drawbacks.

2.1 Ubiquitous Mobile Devices

The world has seen rapid growth in the availability of mobile devices in recent years. It is predicted that the overall number of mobile devices will reach 5.6 billion, and more than 90% of adults in developed countries will own at least one mobile device by 2023 [94]. The umbrella term 'mobile device' refers to a wide range of portable devices equipped with computing power, storage capacities, and wireless network access. The most common mobile devices are smartphones and tablets, capable of hosting a broad range of applications in various areas such as education, health-care, and entertainment. In recent years, we also have witnessed a new trend in wearable mobile devices, which are worn close to or on the surface of users' skin. Wearables are usually designed to help users perform useful micro-tasks, such as checking incoming text messages or monitoring users' physiological data [114]. More recently, a new type of mobile device called e-paper is gain in popularity [137]. E-paper devices typically refer to devices equipped with electrophoretic screens (i.e., e-ink). Compared with other mobile devices, E-paper devices possess several essential advantages. Firstly, they need no power to hold static content on the screen. It only consumes a small amount of energy for updating the contents. Thus, E-paper devices tend to have long

battery life. Secondly, E-paper provides a unique user experience for viewers due to its stable image, wide viewing angle, and high readability. Lastly, E-paper devices can also have a flexible or rollable form factor, which is valuable to products that demand high portability. E-paper devices have been applied in numerous useful applications across various domains, such as e-book readers, electronic shelf labels, and indoor displays.

Note that there exist several umbrella terms closely related to mobile devices, including Internet of Things [77], Smart devices [103], or Edge devices [2]. The resemblance among them is the ability to conduct computation, store data, and connect to the Internet. In this thesis, we may use these terms interchangeably.

2.2 Native Development

Traditionally, mobile applications are developed using programming languages and development environments designed for specified mobile platforms. This type of development is typically referred to as the native development approach. An inherent consequence of the native development approach is that if an app wishes to reach audiences across multiple platforms, the app has to be written multiple times. For example, once for Android using Android Studio and Java, Kotlin, or C++, and a second time for iOS using Xcode and Objective-C or Swift [49]. With more and more mobile platforms emerging on the market, the efforts required to support them are strenuous. Device fragmentation issues further worsen this problem. Take the Android platform as an example. Nowadays, Android exists not only in many versions but with numerous vendor-specific changes. Each Android device may provide a different set of operating system APIs. It is thus infeasible for Android developers to test their apps exhaustively. There is no guarantee that an Android app developed in a generally acknowledged way will be executable on the multitude of Android devices that offer compatibility in theory [18]. Another major drawback of the native development approach is that it leads to multiple separate codebases. Maintaining separate codebases is tiresome and prone to errors because each modification now has to be replicated to all platform-specific variants. As a consequence, new updates and bug fixes for

the app now would take several times longer with multiple platforms compared to only one platform [53].

2.3 Cross-platform Development

To address the limitations of the native development approach, many researchers are actively searching for alternative development approaches. In industry and academia, the alternative approaches are usually covered under an umbrella term: cross-platform development [68]. The primary goal of cross-platform development is to achieve 'write once and run everywhere', in other words, to enable the use of a single codebase across multiple platforms while providing acceptable application performance [141].

A majority of research works [68, 17] classify the current cross-platform development approaches into three main categories: Cross-compiled, Model-driven, and Web-based. The following sub-sections briefly introduce the fundamental concepts of each category and analyze its benefits and drawbacks.

2.3.1 Cross-compiled Approach

The cross-compiled development is also referred to as syntax-directed source-to-source translation. Its main idea is to use compilers to translate a codebase written for a specific platform to another platform. Well-known examples include J2ObjC [11] and XmlVM [106], which translate Java for the Android platform into Objective-C for the iOS platform. Another useful tool is Xamarin [99], which enables the conversion from C# for the Window platform to Native byte code executable on iOS or Android.

However, source-to-source translation cannot generally translate platform-specific APIs. API translation is necessarily challenging due to the structural and idiomatic differences in how functionalities are expressed across mobile platforms [22]. To alleviate this issue, researchers have proposed a solution known as code recommendation systems. Code recommendation systems aim to identify a source code block's semantic content and formulate an appropriate query to search for the equivalent target code block using popular web-based

programming resources. For example, Selene [122] recommends equivalent code blocks by searching millions of example programs to provide usage examples for a given input code block. Sourcerer [10] achieves similar functionalities by searching repositories such as Sourceforge or Github. Some research works [102, 22] apply sophisticated machine learning algorithms such as vector space and linear models on live StackOverflow data to achieve a higher level of translation quality.

While useful, none of these tools can guarantee the correctness of translation results. Consequently, developers still need to examine or debug the converted codes manually. This process can be tedious and error-prone.

2.3.2 Model-driven Approach

This approach derives from a well-known software development paradigm called Model-Driven Development (MDD) [53]. The main idea of MDD is to describe a problem in a model and generate software from this representation. In the context of mobile cross-platform development, developers can describe their apps using comparatively high-level languages, commonly referred to as Domain-Specific Languages (DSL). The description is automatically translated into native code for various mobile platforms. Two pioneering examples are Applause [9] and Cabana [37], which provide code generators for iOS, Android, Windows Phone 7, and Google App Engine. Axiom [65] proposed a different DSL, which is based on the programming language Groovy and resembles features of UML. MD2 [53] is another competing DSL, which supports the Model/View/Control (MVC) design pattern.

However, two negative factors hinder the widespread use of model-driven development. Firstly, most existing DSL languages are prototypic and not ready for production usage. They usually lack clear documentation, inevitably makes them hard to learn [75]. The problem is worsened since every model-driven framework develops and exposes its distinct DSL, rendering knowledge transferability from one DSL to another low or non-existent. Secondly, DSLs are infamous for its poor interoperability with other programming languages, meaning it is difficult to integrate or reuse existing software components written in conventional programming languages in a DSL-based app [33].

2.3.3 Web-based Approach

Cross-platform development only began to gain widespread popularity when the web-based approach has come forth. Web-based development enables developers to implement cross-platform mobile apps using web programming languages such as HTML5, CSS3, and Javascript. The Web-based approach possesses a variety of advantages. Firstly, web programming languages are inherently portable from one mobile platform to another thanks to the standardization of browsers. Every mobile browser nowadays adheres to HTML5 standards and guarantees consistent behavior across all platforms. Secondly, modern mobile browsers allow an app to access a rich set of native device features such as geo-localization, multimedia, and storage. It enables developers to implement sophisticated functionalities as they can do in native development. Thirdly, unlike DSLs, Javascript is one of the most widely used programming languages and has a mature and ever-increasing community. Therefore, web developers can have easy access to abundant documents and technical support from the Internet.

Depending on how a web-based app is loaded and rendered, the pool of web-based approaches can be further divided into three categories: *Hybrid*, *Interpreted*, and *Progress Web*.

Hybrid: The hybrid approach works internally by initializing a native app, which includes a WebView component [126]. The component essentially serves as an embeddable web browser. It is in charge of executing and rendering HTML, CSS, and JavaScript files. They make up the app's business logic and user interface. The hybrid approach is also referred to as native-wrapper because it wraps web assets into a publishable and deployable native app [139]. To facilitate the hybrid development, a number of frameworks such as Cordova [133], PhoneGap [5], Ionic [62] and MoSync [90] have been proposed. These frameworks' general goal is to automatically generate a native wrapper and facilitate the packaging of HTML, CSS, and JavaScript files. Apart from that, these frameworks also permit two-way communication between the WebView and native code. Such communication, usually called Foreign Function Interface (FFI), was designed to access native device functionality such as Geo-localization, Bluetooth, and network connectivity from the Javascript context. Nevertheless,

FFI communication may be deprecated shortly because modern browsers already allow an app to access a wide range of native device features via Javascript APIs.

Interpreted: Unlike the hybrid approach that reuses the device's inbuilt browser through a WebView component, apps built with the interpreted approach ship with a self-contained browser runtime (e.g., JavascriptCore [140] for iOS and V8 [125] for Android). The benefits of bundling browser runtimes are two-folded. Firstly, it alleviates compatibility issues in old mobile devices. Take the Android platform as an example. According to the Android distribution dashboard [8], approximately 20% Android users are still running old versions of the Android OS (below 6.0) by the end of July 2020. These users are unlikely to receive any updates from their device vendors. Their WebView components may be too outdated to execute modern HTML5 apps. Shipping a runtime allows these devices to enjoy up-to-date browser functionalities and avoid many security vulnerabilities. Secondly, many customized browser runtimes can render native user interface components to the screen, instead of pure HTML views. It allows developers to display native-like user interface components that conform to the design guidelines of all supported platforms [36], in other words, delivering better user experience.

Exemplary interpreted frameworks includes React Native [85], Titanium Appcelerator [101], and NativeScript [30]. However, one major drawback is that each framework has proposed its runtime environment, featuring different file structures and underlying plugin architectures. Consequently, a codebase developed for one interpreted framework will not work in another framework out of the box.

Progressive Web: Progressive Web Apps (PWA) have increasingly gained popularity amongst practitioners recently. Unlike the previous two approaches that require a download of binary files to users' devices, a PWA is hosted on a webserver and served to users via a web page. When accessing a PWA-enabled web page, the user will be prompted with a banner requesting permission to install the app onto their devices. The installation process will download all web assets into the browser's local storage. An app icon will also be added to the user's home screen, similar to native apps. Upon launching the app from

the home screen, a PWA-optimized browser window will load the previously downloaded assets and start rendering user interfaces. Note that a PWA optimized browser window will only display the app's content while hiding other browser user interfaces such as address bars and settings menus. Unless the user is familiar with PWA's concept, they will not realize that the app is running within a browser. In other words, PWA apps deliver a user experience similar to native apps. Another significant difference to a regular website is that a PWA app can use Background Service Workers, whose purposes are to manage the app's lifecycle, control data synchronization, and receive push notifications even after the browser is closed [45].

Progressive Web Apps is gathering momentum. Recent research convers performance analysis [47], evaluation frameworks [111], energy consumption [27], and user experience [45]. Their contributions serve as an important foundation for future research on PWAs.

2.3.4 Research Challenges in Web-based Approach

The web-based approach has significantly facilitated cross-platform development, yet the following two challenges still await.

Performance. Measuring the performance of apps built with cross-platform development frameworks is always a hot research topic. There is a consensus that cross-platform frameworks potentially leads to decreased performance compared to the native development approach. For example, Mercado et al. [87] leverage secondary data taken from app store reviews to monitor performance issues of apps. They report that cross-platforms apps are more likely to encounter performance issues. Ciman et al. [27] conduct a comprehensive evaluation of resource consumption for popular cross-platform frameworks and observes significant performance overhead compared to native development. Similar studies [107, 16] also indicate that cross-platform frameworks are more computational-intensive than native apps. For example, certain functionalities such as file system access, GPS sensor usage, and video playback can be roughly 20 times as slow as native apps [31]. Some research works [58] even strongly argue that there is 'no choice other than native for performance'; in other words, developers should stick to native development for high app performance.

Several empirical studies [19, 83, 139] suggest that on the one hand, the performance penalty is generally acceptable on high-end devices. On the other hand, it can significantly degrade user experience or even render apps unusable. Some research works report that the culprit of performance degradation is Javascript. Sommer et al. [118] report that JavaScript-based frameworks encounter performance penalties more frequently due to the fact that JavaScript is notoriously difficult to compile and optimize efficiently [12, 113]. Aline et al. [40] and Andreas et al. [18] report a similar finding that applications written in JavaScript run much slower than their native counterparts. It is generally believed that Javascript is not suitable for building demanding applications such as interactive 3D visualization, audio and video software, and games [50].

To date, how to build demanding cross-platform applications for resource-constrained mobile devices remains an open research question.

Interoperability and Reusablity. For a long time, JavaScript seems to be the only available programming language in web-based cross-platform development. Unfortunately, as we mentioned above, Javascript is likely to encounter performance issues. Moreover, Javascript does not easily interoperate with other languages. It renders running other programming languages a challenging task. There exist reasonable motivations for running code written in other programming languages. For example, a software company may wish to reuse a large amount of existing code written in another language. A developer may simply have a strong preference for another language or is more productive in it.

As a workaround, there exist tools to convert different programming languages to Javascript. For example, Google Web Toolkit [35] aims to convert Java to Javascript. SCM2JS [82] is a compiler that translates a variant of the Scheme programming language into JavaScript. AFAX [98] compiles F# into Javascript. Similarly, Links [29] is designed to translate meta languages to Javascript. While useful, such tools usually only allow a subset of the source languages to be converted. For example, it is generally hard to translate multi-threaded code to Javascript since Javascript is asynchronous and does not have a direct parallel to threads. Also, there may exist a semantics mismatch between the source language and the target language. The mismatch may be

too subtle to be observed and challenging to be bridged. An example is that the statement $A = 3/2$ has different semantics in Python and Javascript. In Python, it should yield an integer 1, but in JavaScript, a floating-point number 1.5 is generated. To convert the statement from Python to Javascript, one has to use bitwise operators ($A =\sim (3/2)$). Some research works [50] thus argue not to treat JavaScript as a compilation target due to its many semantic pitfalls and inconsistent performance.

Until now, how to integrate or reuse general programming languages on the web-based apps remains challenging.

2.3.5 WebAssembly: A potential solution?

The recent rise of the WebAssembly has shed some light on the two challenges. WebAssembly is a low-level programming language supported by approximately 95% of global browser installations [135, 78]. Musch et al. [92] report that 1 out of 600 sites among the Alexa 1 million leverage WebAssembly by 2019. One-third of these websites even spend more than 75% of CPU time executing WebAssembly. The popularity of WebAssembly is mainly attributed to the following factors.

Compact Binary Format. WebAssembly defines a binary format, which is compact, statically typed, and fast to parse. To allow WebAssembly to be read and edited by humans, there is a textual representation of the WebAssembly binary format. A sample program in textual representation is demonstrated in Listing 2.1. In common practice, a WebAssembly file will contain a WebAssembly module. A module can contain functions, globals, one linear memory region, and an indirect call table. Unlike most native binary formats running on register-based machines, WebAssembly is executed on a stack-based virtual machine. Specifically, each WebAssembly instruction pops its inputs from and push its results to the implicit value stack. WebAssembly also does not use registers. Instead, WebAssembly can store values in an unlimited number of global variables, whose scope is the entire module. Alternatively, WebAssembly can use local variables, which are only visible to the current function. WebAssembly has been designed to serve as a universal compiler target from the very beginning. As a result, there exists a variety of compiler toolchains (e.g., C++ [143], Rust

```
(module
 ;; Import function from host environment.
 (import "print" (func $print (param i32)))
 ;; Global variable, 32-bit integer, initialized to 42.
 (global $g i32 (i32.const 42))
 ;; Function in the binary with type [i32] -> [i64].
 (func $f (param $arg i32) (result i64)
    (local $var i32) ;; Declaration of a local variable.
    i32.const 8 ;; Push constant on stack.
    local.get $arg ;; Copy function argument to stack.
    i32.add ;; Pop inputs from stack, push result.
    local.tee $var ;; Copy result to local variable.
    if ;; Is top == 0?
    i32.const 1024 ;; Pointer to string in memory.
    call $print ;; Call imported function.
    end ;; Structured control-flow.
    local.get $var ;; Push local value as address
    i64.load ;; ...8 byte read from linear memory.
 )
 ;; Explicitly initialized memory at offset 1024.
 (data (i32.const 1024) "some string\00")
)
```

Listing 2.1: Example of a WebAssembly binary, represented in the text format, adapted from [78]

[86] and Go [100]) that target WebAssembly. Codes written in these languages can be compiled to WebAssembly and executed in browsers, provided that they do not use system abstractions.

High Performance. WebAssembly delivers significantly higher performance than Javascript due to the following design choices. Firstly, WebAssembly binaries are smaller than textual JavaScript files, and thus are faster to download. It is especially beneficial for slow networks. Secondly, Javascript features a long startup time because the javascript text needs to be transformed into a data structure named Abstract Syntax Tree (AST) before execution. On the contrary, WebAssembly is delivered as binary and skips the time-consuming tree parsing process. Thirdly, distinct to dynamically typed Javascript, WebAssembly is statically typed. As a result, the WebAssembly runtime does not need to speculate about variable types during compilation. This makes native code generation more efficient for WebAssembly.

Type Checking. Unlike in most native binary formats, WebAssembly assigns type information to instructions, global/local variables, and functions' arguments and results. WebAssembly runtimes can verify the type information during validation, instantiation, and possibly execution. There exist four primitive types, including i32 (32-bit integer), i64 (64-bit integer), f32 (single-precision float), and f64 (double precision float). More complex types are lowered to these primitive types by compilers. Type checking ensures that a WebAssembly module is meaningful and safe.

Structured Control Flow. Very distinct to native code or Java bytecode, WebAssembly features a structured control flow. In detail, WebAssembly instructions are organized into well-nested blocks, and branches can only jump to the end of the surrounding blocks. It is strictly forbidden to use unrestricted goto statements or jumps to arbitrary addresses. Moreover, WebAssembly uses separate areas to store instructions and data; in other words, one cannot execute data as instructions. As a result, many conventional vulnerabilities are thus ruled out in WebAssembly, such as shellcode injection or abusing unrestricted indirect jumps.

Due to the missing of goto statements in the structured control flow, WebAssembly introduces a special instruction *call_indirect* and a *Table* section to emulate function pointers and virtual functions. The call_indirect instruction pops a value from the value stack and uses it as an index to the table. Each entry of the table corresponds to the memory address of a function, which is subsequently called. In WebAssembly, a function can only be indirectly called if it is present in the table section.

Linear Memory. Unlike Javascript, WebAssembly does not provide a managed memory or garbage collection. Instead, WebAssembly adopts a straightforward linear memory, which is a single dimension array of bytes. WebAssembly can use *Load* and *Store* instructions to read/write contents within the linear memory. The memory is addressed by a 32-bit integer (i32). As a result, A WebAssembly binary can support a maximum of 4GB memory. Most WebAssembly runtimes may only allocate a small chunk of linear memory (e.g., 16 MBs) before execution. The program can request the runtime to increase the linear memory with the *memory.grow* instruction. To efficiently manage the linear memory, a

WebAssembly binary typically includes its own memory allocator (e.g., using Doug Lea memory allocator [76] to provide *malloc* and *free* functions).

System Calls. A WebAssembly runtime will typically be embedded into a host environment. An embedder implements the necessary connections between such a host environment and the WebAssembly runtime. Not surprisingly, most WebAssembly runtimes are embedded in Web browsers since WebAssembly was initially designed to run on the Web. In browsers, all APIs (e.g., Ajax Request or DOM manipulation) available to JavaScript engines will be imported to WebAssembly runtimes as system calls. Recently, other non-web environments are also emerging. For example, Node.js [125] now allows the execution of server-side WebAssembly applications. Node.js has full control over what APIs should be exposed to WebAssembly modules. For example, Node.js can refuse to expose any APIs to WebAssembly such that only pure computation is allowed. Node.js can also reveal all native system calls, and a WebAssembly then can invoke exec to execute any shell commands.

Recently, an effort has been made to standardize the system interfaces between the host environment and the WebAssembly runtime. WebAssembly System Interface (WASI) is a standardization for a modular system interface for WebAssembly supervised by a W3C subgroup. It defines a rich set of POSIX-like APIs that a runtime should implement to run WebAssembly modules as standalone applications. Another ongoing work is the WebAssembly Interface Types, which allow WebAssembly modules to interoperate with abstract types from other languages using a simple interface. For now, WebAssembly can only communicate with other languages via memory sharing; non-primitive data such as strings or objects must be passed through linear memory.

Precursors to WebAssembly. Before WebAssembly, there have been several attempts to execute native codes in browsers.

ActiveX [104] is a deprecated technology that allows web pages to load and execute signed Windows x86 libraries. However, this opened a security loophole since these binaries have unrestricted access to the Windows API. This issue does not impact WebAssembly since each WebAssembly binary is executed within a sandboxed environment separated from the host runtime using fault isolation techniques.

Google Chrome also introduced Native Client (NaCl), a sandbox for running C++ code in the browser efficiently and securely [142]. Because NaCl relies on static validation of machine code, it requires compilers to follow specific patterns and only supports a small subset of the x86 and ARM instructions. To address the inherent portability issue, Google Chrome later introduced Portable NaCl [39], which uses LLVM Bitcode as a binary format. However, LLVM bitcode is not always portable because it still exposes platform-specific details such as the call stack layout. Google later abandoned both technologies in favor of WebAssembly.

Another attempt is Asm.js [55], which is a newly introduced subset of JavaScript designed to be compiled to native code. Asm.js tries to avoid the dynamic type system of JavaScript by introducing type coercions. Asm.js has no inbuilt support for 64-bit integers. Emulating them in Javascript can cause significant performance overhead. Asm.js is being abandoned in favor of WebAssembly since WebAssembly has been shown to provide much better performance [64].

2.3.6 Missing Pieces of WebAssembly

It would seem that WebAssembly is what we all need to solve the performance and interoperability issues in web-based cross-platform development. However, in reality, there are still missing pieces of the puzzle.

Despite WebAssembly's promise of native performance, applications compiled to WebAssembly still run slower by an average of 45% as reported by previous benchmarks [64]. The major causes of the slowdown are two-folded. Firstly, conventional WebAssembly runtimes translate WebAssembly instructions to native machine code using just-in-time compilers, which do not apply complex code optimization. The generated binaries may suffer from poor instruction selection, excessive register spills, redundant branch statements, and unnecessary L1 instruction cache miss [64]. Secondly, system calls from WebAssembly applications have to be proxy-ed by runtimes to reach operating systems. For kernel-intensive applications, the proxying mechanism can incur a significant performance overhead.

In terms of interoperability, even though many compilers can now target WebAssembly, the compilation can still fall short in practice because it overlooks an important fact. Most popular programming languages and their standard libraries expect abstractions from modern OSs such as threads, file system, and sockets. These abstractions are not directly available and can be hard to emulate in browsers. Consequently, existing codebases still have to go through substantial modifications to accommodate the missing abstraction problem. Some research work aims to address these issues. Doppio [131] emulates a small range of runtime and operating system abstractions using Javascript in the browser environment to execute Java Virtual Machines. Browsix [105] mimics a Unix kernel within the browser and includes a compiler to compiles native programs to JavaScript. Together, they allow native programs (in C, C++, and Go) to run in the browser and use most operating system services, such as filesystem, processes, and pipes. However, these compatibility layers may not provide the same behaviors as native operating systems do. For example, none of these can emulate prioritized threaded execution inside browsers. This inconsistency may result in unexpected program errors. Moreover, this approach would not work if the developers do not have access to source code but the binary files.

Until now, the academic body of knowledge on WebAssembly is limited. Therefore, it is meaningful to carry out more research involving technical innovation, conceptual discussions, and case studies of WebAssembly, especially within web-based cross-platform development.

Chapter 3

BrowserVM: Running Unmodified Operating Systems and Applications in Browsers

Web browsers are becoming a de-facto universal computing platform and exist in nearly every mobile device. Web-based cross-platform development leverages the ubiquity and enables developers to implement portable apps using Javascript and WebAssembly.

Recently, research communities are attempting to enhance browsers to run applications written in general programming languages. Existing approaches mainly compile source codes to browsers' native instruction sets JavaScript or WebAssembly. However, they usually fall short in practice because browsers lack operating system abstractions (e.g., thread and filesystem), and many programs would require extensive modifications. This chapter presents BroswerVM, a new approach to run unmodified and complete operating systems and applications inside browsers. BrowserVM is a WebAssembly-based virtual machine hypervisor. BrowserVM efficiently conducts processor emulation through dynamic binary translation. It also provides performant hardware emulation for hard disks, graphics cards, and network adapters. Implementing BrowserVM is challenging because of the unique characteristics of WebAssembly: semantic gap with low-level CPU assembly and high initialization overhead. We detail the methods to deal with these challenges and conduct a performance benchmark on BrowserVM.

Our results indicate that though slower than native hypervisors, BroswerVM provides acceptable performance to execute existing applications that are not compute-intensive.

3.1 Rationale

Nowadays, web browsers have become ubiquitous; virtually every computing device, from desktops to mobile phones, comes with an inbuilt web browser. Browsers are also getting increasingly faster. They now incorporate highly-efficient compilers for JavaScript and support WebAssembly (an assembly-like language that runs with near-native performance). These merits make it possible for modern browsers to deliver cross-platform and compute-intensive applications.

More recently, many research projects attempt to add support to browsers for more programming languages other than JavaScript and WebAssembly. This support is strategically important because it makes browsers a universal computing platform able to execute a large body of existing applications. Most attempts achieve this by designing a toolchain to compile source codes to Javascript and WebAssembly. However, this compilation approach falls short in practice as it overlooks an important fact; most popular programming languages and their standard libraries expect abstractions from modern OSs such as threads, filesystems, and sockets. These abstractions are not directly available and can be hard to emulate in browsers. As a consequence, existing codebases have to go through substantial modifications to accommodate the missing abstraction problem. Moreover, this approach will not work if the developers do not have access to source code.

We present BrowserVM, a new approach to run general programming languages in browsers to address the issues. BrowserVM emulates a CPU and peripheral devices (e.g., hard disks or graphics cards) to execute any general-purpose OSs and applications inside browsers. Unlike the compilation approach, BrowserVM requires neither compilation nor modification of existing software. It guarantees binary compatibility; disk images prepared for conventional virtual machines will also work on BrowserVM out of the box.

A critical component of BrowserVM is CPU emulation, which enables a source instruction set A to execute on a processor supporting a target instruction set B. In our context, the source instruction A set can be *X86/64* (used in desktop computers) or *AArch64* (used in mobile devices). The target instruction set B is in *Javascript* or *WebAssembly*. In this chapter, WebAssembly is chosen for its near-native execution performance. An efficient approach for CPU emulation is dynamic binary translation (DBT), where sequences of source instructions are translated to target instructions at runtime. However, we note that implementing DBT inside browsers has many unique challenges. Firstly, conventional DBT is designed to translate a low-level instruction set to another low-level instruction set with similar semantics. Our translation target WebAssembly is relatively high-level and lacks a commonly used low-level semantics (goto statements). The semantic mismatch makes translating instructions less straightforward. Secondly, unlike other low-level instruction sets, the translated WebAssembly binary is not directly executable by physical CPUs. It has to be fed into a WebAssembly compiler to generate native machine codes before execution. This initialization step is CPU intensive and memory intensive. If it is frequently invoked as is done in conventional DBT, the overhead can outweigh the performance benefits of WebAssembly. How to suppress the initialization costs remains an open challenge.

In the following sections, we present practical solutions to cope with the above challenges. To resolve the semantic mismatch, we apply a control flow graph transformation to replace goto statements in source instructions with multilevel break statements in WebAssembly. To reduce the WebAssembly compilation overhead, we propose a generational execution scheme to suppress unnecessary compilation and improve memory utilization. As most translation works are repetitive among different runs of an executable, we propose a translation cache reuse mechanism to reduce the emulation costs further. In practice, many applications are not only compute-intensive but also I/O or graphics-intensive. BrowserVM optimizes their performance by providing efficient emulation of hard disks, network adapters, and graphics cards.

We consolidate the above techniques and implement a prototype system for mainstream browsers. We benchmark the prototype and provide performance

measurements on the CPU emulator. The experiment results show that while slower than a conventional virtual machine, BrowserVM provides acceptable performance for non-compute-intensive tasks.

The contributions can be summarized as follows:

1. We describe how to implement CPU emulators in browsers using dynamic binary translation from low-level CPU instructions to high-level WebAssembly in Section 3.3.

2. We present a multi-tier execution scheme and a translation cache reuse mechanism in Section 3.4 to minimize the initialization overhead of WebAssembly.

3. We illustrate how to emulate essential hardware, including hard disks, graphics cards, and network adapters in Section 3.6.

4. We conduct a quantitative performance evaluation on our prototype in Section 4.5.

3.2 Background

Modern web browsers are becoming an increasingly popular computing platform. Its instruction sets Javascript and WebAssembly enable developers to deliver portable executables with near-native performance. Recently, researchers are adding more language support to browsers. It can grant browser application developers the freedom to use their favorite programming languages and paradigms (e.g., functional or object-oriented). It also allows reusing a large body of existing well-tested codebases, vastly speeding up application development and reducing the risk of introducing errors.

Source to source compilers: One common approach is to design compilers that translate conventional languages to WebAssembly or Javascript. Examples include GWT (Java to Javascript) [120], Emscripten (C to WebAssembly) [143], IL2JS (.Net to Javascript) [61], and Rust-lang (Rust to WebAssembly) [86]. However, in practice, this approach usually falls short due to two reasons. Firstly, this approach requires access to source code, which is

not always possible. Secondly, conventional programming languages and their standard libraries expect a rich execution environment, such as one provided by modern operating systems. Browsers lack many standard operating systems abstractions (e.g., thread and filesystem). It means that existing source code can not be run without substantial modifications. For example, mainstream browsers solely support single-threaded event-driven execution and have no support for interrupts. Events are either executed to completion or killed when the execution time limit is reached. This runtime model forces developers to refactor multi-threading applications in an asynchronous event-driven paradigm. This process can be tedious and bug-inducing. A few papers such as Doppio [131] and Browsix [105] aim to address these issues. The authors decided to emulate a small range of runtime and operating system abstractions using Javascript. However, these compatibility layers may not provide the same behaviors as native operating systems do. For example, none of these can emulate prioritized threaded execution inside browsers. This inconsistency may result in unexpected program errors.

Emulators in Browsers: BrowserVM presents a new approach to the problem. The main idea is to emulate a CPU that directly executes binary codes inside browsers. BrowserVM requires no access to source codes. To date, there are two common implementation approaches for CPU emulation: *Interpretation* or *Dynamic Binary Translation* (DBT). Interpretation is relatively simple to implement. JSLinux [13], jor1k [66] and v86 [128] have experimented with this approach inside browsers. Its core mechanism is summarized in Algorithm 1. The interpreter steps through the source program one instruction at a time, which is then decoded and dispatched to the corresponding interpreter routines. This read-decode-dispatch loop is inefficient because it incurs a high branching overhead for each instruction (of at least five branches).

The alternative approach DBT is shown in Algorithm 2. DBT divides and translates source binary into a set of basic blocks. A basic block is defined as a sequence of non-branching instructions following a branch instruction. By executing the binary at a block level, DBT has a significantly smaller branching overhead than interpretation. A translated block is also cached so that it can be executed repeatedly at almost zero cost if the program branches to it again.

Algorithm 1: Pseudo code of an interpreter

```
while not halt and not interrupt do
    increase(PC)                          // Increase program counter.
    inst = code[PC]  // Fetch an instr
    opcode = decode(inst)  // Decode the instr
    switch opcode do
        // Dispatch to corresponding routines
        case Branch do
            Branch(inst)  // Handle branch instrs
        case ALU do
            ALU(inst)  // Handle arithmetic instrs
        ...
```

Algorithm 2: Simplified Pseudo Code of DBT

```
while not halt and not interrupt do
    blockAddr = cacheLookup(PC)                  // Check Cache
    if not blockAddr then
        blockAddr = translateBlock(PC)           // Translate
        cacheBlock(PC, blockAddr)                // Cache result
    execBlock(blockAddr)                         // Execute the block
    updatePC(PC)                                 // Update the program counter
```

It should be noted that the DBT algorithm we show here is simplified and not fully optimized. It looks up the block cache after executing each block. To improve this, we can leverage a fact that during execution a block is very likely to be followed by a deterministic successor block. As a result, we can chain the two blocks and execute them together to reduce the number of cache lookups. The chaining is usually achieved by placing a goto instruction (jump to an address) at the end of a translated block.

DBT outperforms interpretation by one order of magnitude in terms of execution speed, as we will see in Section 4.5. However, at the time of writing, few DBT implementations for browsers exist. This scarcity mainly stems from the unique characteristics of WebAssembly, which make browsers not directly suitable for CPU emulation. We will illustrate the challenges imposed by those characteristics and provide workable solutions in the following sections. To the best of our knowledge, BrowserVM is the first research work to experiment with DBT-based CPU emulation in browsers.

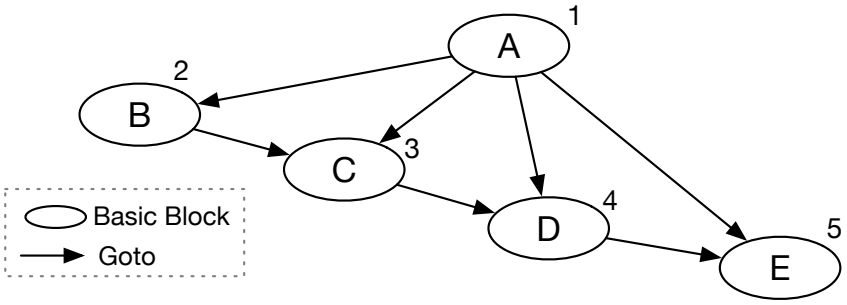


Figure 3.1: An example of unstructured control flow, where basic blocks are connected with branching edges. Numbers reflect the topological order of each block.

3.3 Binary Translation to WebAssembly

The first technical challenge in implementing DBT in browsers is to translate low-level CPU assembly (e.g., X86 or ARM) to high-level WebAssembly. The translation is generally straightforward for arithmetic instructions since most of them have corresponding substitutions in WebAssembly. Table 3.1 demonstrates some simplified conversion examples. The tricky part lies in the transfer of control instructions; most CPU instructions feature *unstructured control flow*, while WebAssembly follows *structured control flow*. Unstructured control flow, as shown in Figure 3.1 and Listing 3.1, represents code as a directed graph with basic blocks as nodes, and branching instructions as edges. Unstructured control flow usually allows the use of goto statements to enable the flexible transfer of control among all basic blocks. In contrast, structured control flow, as illustrated in Listing 3.2 and Listing 3.3, represents code as an ordered sequence of basic blocks surrounded with high-level control flow constructs (e.g., loop/block, if/else and break/continue). Notably, it does not include any form of goto statements, and the transfer of control can only occur from a block to its surrounding blocks.

Despite the semantic differences, we know the translation is always feasible because WebAssembly and CPU assembly are both Turing-compete and can precisely emulate each other. In fact, we can utilize a simple approach called *Relooper*. Its key idea is to construct a state machine that branches to the

```
// C program.
int main(int argc, char **argv)
{ //line 10
  switch(argc)
  {
    case 1:
      square(1);
      break;
    case 2:
      square(2);
      break;
    default:
      square(3);
  }
  return 0;
}

// Corresponding unstructured X86 Assembly
main:
        ...
        je .L4
        cmp DWORD PTR [rbp-4], 2
        je .L5
        jmp .L9 // Goto statement
.L4:
        mov edi, 1
        call square(int)
        jmp .L7 // Goto statement
.L5:
        mov edi, 2
        call square(int)
        jmp .L7 // Goto statement
.L9:
        mov edi, 3
        call square(int)
.L7:
        ...
```

Listing 3.1: Another example of unstructured control flow in the assembly form. Notice that the 'jmp' statements enable transfer of control to any lines.

Table 3.1: X86 arithmetic instructions V.S. WebAssembly arithmetic instructions

Instruction Description	X86	WebAssembly

Integer Add	mov eax, opa
	mov ecx, opb
	add eax, ecx
		i32.const opa
		i32.const opb
		i32.add

Integer Subtract	mov eax, opa
	mov ecx, opb
	sub eax, ecx
		i32.const opa
		i32.const opb
		i32.sub

Multiply	mov eax, opa
	mov ecx, opb
	imul eax, ecx
		i32.const opa
		i32.const opb
		i32.mul

Performs bitwise logical AND	mov eax opa mov ebx, opb and eax, ebx	i32.const opa i32.const opb i32.and
Performs bitwise logical OR	mov eax opa mov ebx, opb or eax, ebx	i32.const opa i32.const opb i32.or

Shifts arithmetic right	mov eax, opa
	mov ecx, opb
	sar eax, cl
		i32.const opa
		i32.const opb
		i32.shr_s

right block with the help of a conditional variable (state label). Listing 3.2 showcases the generated state machine for our example in Figure 3.1. This solution is inefficient because it incurs a significant overhead due to the frequent writing, reading, and branching on the conditional variable. To alleviate this performance issue, Relooper incorporates a greedy optimization process: it tries to recognize some common patterns in unstructured control flow and replace them with readily optimized high-level constructs. If no pattern is matched, the algorithm falls back to the state machine approach. This process can usually remove most redundant access to the conditional variable. However, given its greedy nature, the algorithm may still produce sub-optimal code. The example we have shown above is one case where no label access can be factored out.

BrowserVM adopts a more efficient approach by taking advantage of a special construct in WebAssembly: multilevel break. This statement is designed to enable the transfer of control to enclosing loops at any level (i.e., break

```
let label='A';
loop {
  if (label == 'A') {
    A();
    if (A_to_B) label='B';
    elif (A_to_C) label='C';
    elif (A_to_D) label='D';
    else label='E';
  } elif(label == 'B') {
    B(); label='C';
  } elif (label == 'C') {
    C(); label='D';
  } elif (label == 'C') {
    D(); label='E';
  } else {
    E();
  }
}
```

Listing 3.2: Using Relooper algorithm

out of nested loops). It allows us to use an important finding in programming language theory: an unstructured control flow graph can be transformed to a structured form only consisting of loops, conditional branches, and multilevel break statements [108]. Without going into in-depth details such as irreducible loops handling, we can summarize the main transformation steps as follows. Firstly, reorder and output basic blocks in topological ordering: if two basic blocks X and Y are connected with a forward edge $X \rightarrow Y$, X comes before Y in the ordering. Secondly, examine all the forward edges in the ordering. If the source and destination blocks are consecutive in the topological order, nothing needs to be done. Otherwise, we need to insert a loop scope somewhere in the output, such that 1) the destination block sits just after the end of the new loop scope, and 2) the loop scope starts at the beginning of the output. The last step is to insert multilevel break statements branching to the newly created loop scopes to achieve control transfer. Listing 3.3 demonstrates the transformation result. Compared with Relooper, it does not generate extra conditional variables and is likely to deliver higher execution performance.

It is worthwhile to mention that BrowserVM reuses and benefits from the

```
loopE:{
  loopD:{
    loopC:{
      A();
      if (A_to_C)
        break loopC;
      if (A_to_D)
        break loopD;
      if (A_to_E)
        break loopE;
      B();
      break loopC;
    };
    C(); break loopD;
  };
  D(); break loopE;
}
E();
```

Listing 3.3: Using break constructs

QEMU virtualization framework. The QEMU framework provides an intermediate representation layer (IR) between source and target languages. It allows us to focus on translating the abstract IR bytecode to WebAssembly. Our implementation is fully independent of the source assembly. Support for new CPU types can be easily added without modifying our translator.

3.4 Optimizing WebAssembly Execution

To execute the translated WebAssembly binary in browsers, we first need to load and compile it through a series of JavaScript APIs.

The recipe is summarized as below.

1. Fill a WebAssembly binary into an array buffer.
2. Feed the buffer to the Javascript API *WebAssembly.compile* for compilation and obtain a runtime context object.
3. Provide the runtime context with a memory region and necessary functions to handle I/O operations and traps.

4. Invoke the desired functions in the WebAssembly binary.

This initialization process is generally considered to be memory and computationally intensive. Unfortunately, in conventional DBT, the process has to be frequently invoked for every newly translated block, resulting in a considerable amount of overhead.

To alleviate the overhead, BrowserVM attempts to minimize unnecessary WebAssembly compilation. BrowserVM adopts a core concept of just-in-time (JIT) compilers: if a translated block is going to be executed for a small number of times, it is not worth paying the compilation cost. Instead, just interpreting the block may consume fewer CPU cycles. Following this logic, a basic block is initially executed using an interpreter, and only after a certain number T_{jit} of invocations, the block is considered hot and queued for compilation. The hot block threshold T_{jit} is set to 1500, a heuristic number adopted by various industry JIT compilers (e.g., Java virtual machine).

However, even the above strategy could result in thousands of hot blocks at a single time. For instance, our measurements show that a Busybox Linux has approximately 1230 hot blocks after a cold boot. Compiling and managing such a large number of hot blocks is still relatively difficult. A simple management policy is **one-context-one-block**, where we compile each hot block separately and maintain its corresponding context object. However, this policy may waste a considerable amount of memory. For example, in a recent Firefox (v72) version, each context object takes up approximately 132 kilobytes of memory space. Storing thousands of them would require hundreds of megabytes of memory. Nevertheless, on the positive side, this policy allows us to to add or modify a hot block without affecting other blocks. An opposite policy is **one-context-all-blocks**. Under this policy, we merge all hot blocks into a gigantic block for compilation. Therefore, we only need to maintain one context object in memory and can significantly reduce memory pressure. Compared to the former, this policy has an extensive CPU overhead for any changes in hot blocks due to the lack of incremental compilation support in WebAssembly. Every time a hot block is added or modified, we have to re-merge and recompile all the other blocks even if they remain unmodified. Moreover, the overhead increases proportionately to the number of hot blocks. Such a policy would be undesirable

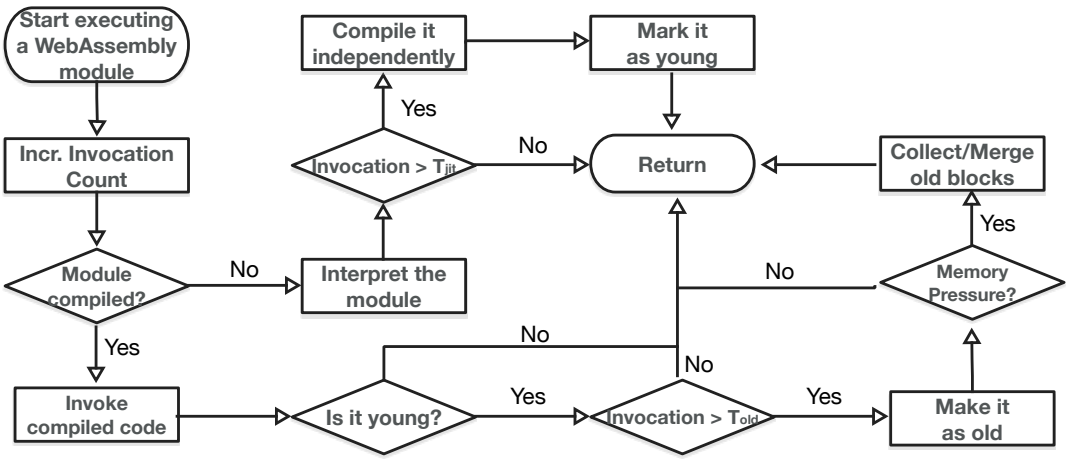


Figure 3.2: The work flow of BrowserVM's generational block management scheme.

for certain types of programs, which continuously generate and modify code segments (e.g., Javascript engines).

To locate a sweet spot between the two policies, BroswerVM uses a heuristic management scheme, as shown in Figure 3.2. Aside from the just-in-time compilation, it exploits two empirically observed properties on how often a block is modified:

1. Young blocks may die young: we define a hot block to be young when it is newly discovered (i.e., the moment it switches from interpretation to compilation). We observe that a young block is more likely to be modified or removed. Therefore, it is reasonable to manage a young block in an one-context-one-block fashion in case of any imminent changes.

2. Old blocks age well: We promote a young block to be an old block after it survives a certain number of invocations T_{old}. We set T_{old} to be 10000 from our empirical measurements. We observe that an old block tends to stay unchanged until the application exits. If we have accumulated a predefined number (e.g., 50) of old blocks in memory, we can merge them into a single block to relieve memory pressure.

3.5 Cache sharing among different runs

In the approach mentioned above, we spend a considerable amount of effort translating and managing an executable's hot blocks. If we can reuse the translation cache, we can skip the hard work in future executable runs.

However, reusing translation cache from previous runs is not straightforward. The existing translation cache uses the physical memory address of a basic block as the cache key, but the address is not likely to be identical across different runs due to various uncontrollable factors (e.g., current memory usage or startup order of applications). To address this issue, we redesign the cache mechanism to use the hash value of a basic block's instructions as the cache key. As long as the code of a basic block is not modified, its key is consistent across different runs. As we mentioned earlier, our implementation is built on top of the QEMU framework, where an IR layer is used. The hash key is computed over IR bytecode using an efficient non-cryptographic 64-bit hash algorithm. We also note that QEMU can often generate more optimized IR bytecode based on the runtime context. For example, on X86 if the SS, DS and ES segments have a zero base, then QEMU can omit addition operations for memory segmentation in IR bytecode. However, overusing this optimization renders cache unsharable because it makes the hash value also dependent on runtime context. Thus, we disable this optimization and argue that it does not negatively impact system performance. The main reason is that the generated WebAssembly binary has to go through a WebAssembly compiler's optimization pipeline, where redundant operations will eventually be factored out.

The cache sharing mechanism sheds light on a more aggressive approach. We may translate an entire application before it is launched so that nearly zero emulation overhead is induced at runtime. This approach is essentially static binary translation, which can be tricky to implement for CISC instruction sets (e.g., X86) due to the infamous code discovery problem. Specifically, CISC has variable-length instructions and allows code interspersed with data or zero-padding bytes. These features easily confuse a translator and make it hard to tell what is data and what is code. As a result, static binary translation is only applicable for certain RISC instruction sets (e.g., RISC-V) designed to have fixed-width instructions and no data mixed with code.

3.6 Hardware Emulation

To support full system emulation, where a complete and unmodified operating system can be run, BrowserVM provides emulation for commonly-used hardware, including hard disks, graphics cards, and network adapters.

In the following sections, we will identify unique challenges imposed by the browser environment on hardware emulation and provide possible solutions.

3.6.1 Hard Disks

Conventional hypervisors emulate a hard disk in a straightforward manner; each emulated hard disk is attached to a file object (i.e., disk image) of the hypervisor, and every disk operation is translated to a read/write operation on the file. However, this approach is not directly applicable in browsers due to the lack of file system APIs. To cope with this constraint, we need to simulate an in-memory file object and necessary I/O operations using Javascript typed arrays. To make the in-memory files persistent, we can serialize and store them using IndexedDB, a Javascript-based object-oriented database.

We provide a wide range of virtual disk images in a web server. A user can fetch a favorite one to play around its prepared application environment instantly. One naive fetching strategy is to download the whole image before launching the emulator. One downside to this strategy is that the loading time may be considerably long. Most applications only access parts of its code and data segments at a single time due to the principle of locality. Therefore, a more reasonable strategy is lazy loading, where we divide the entire disk image into blocks and fetch them on demand. However, this leads to another technical issue: high I/O latency. A block download is accomplished through a HTTP GET request in a browser environment. Its round trip time, depending on the server locations, can range from several milliseconds to hundreds of milliseconds. The latency is significantly high as if the guest was running on a failing hard disk with a long seek time. It will limit system performance since a considerable amount of CPU time is now wasted to wait on the completion of HTTP requests.

In BrowserVM, we present a practical approach to combat the latency issue. The main idea is to speculatively read (i.e., predict and prefetch) multiple blocks

in one HTTP round trip. In this way, the network latency can be amortized over multiple blocks. The detail of this protocol is summarized as follows.

1. To establish a connection to the disk image server, a client first has to send a handshake command, containing a set of block IDs C that the client has downloaded in previous sessions. This information can prevent the server from sending duplicated blocks.

2. When the connection is ready, the client can then send a fetch command to request a specific block b_r. Based on the request, the server predicts a set of related blocks P that are likely to be accessed in the near future.

3. The server then responds to the client with a set of blocks $R = \{b_i | b_i \in b_r \cup P - C\}$. In plain text, the response includes the requested block b_r and the predicted blocks P but excludes the previously downloaded blocks C. Meanwhile, the server also has to update the set C to be $C \cup R$.

The effectiveness of such a protocol is mainly determined by accurate prediction on future block accesses P. One prediction heuristic is to prefetch the next N (e.g., 32 or 16) sequential blocks of b_r using the principle of locality. This approach is very simple but we may sacrifice some accuracy. BrowserVM experiments with a more sophisticated prediction algorithm that takes the block access history into consideration. To formulate our prediction problem, we first provide several definitions. We define $B = \{b_1, ..., b_M\}$ to be a set of disk blocks. We also let $L = \{(t_1, e_1), ..., (t_N, e_N)\}$ to be a stream of block access events in a chronological order, where $e \in B$ and t represents the corresponding timestamp. To facilitate data processing, we partition the events into transactions by timestamp proximity. This partition can be achieved by either chopping the event stream into non-overlapping intervals with the same length or applying the K-means clustering algorithm. We can now redefine $L = \{g_1, g_2, ..., g_K\}$ to be a stream of transactions in a sequential order, where g is a subset of B. We also define an item set x to be a subset of B. A transaction g is said to contain the set x if $x \subseteq g$. The support of x is then defined as the fraction of transactions in L that contain x. The item set x is said to be frequent if the support of x is greater than or equal to a predefined threshold s. With these

definitions, the prediction problem can be formulated as follows. Given B, L and s, the objective is to discover a set of frequent item sets $O = \{x | x \text{ is frequent}\}$. Upon arrival of a block request b_r, we then can quickly identify blocks that are likely to be accessed in near future as $P = \{\cup x_i | x_i \in O, b_r \in x_i\}$.

Finding an exact answer to this problem can be challenging, considering the combinatorial explosion of itemsets in B. To circumvent this issue, we apply an approximation algorithm called *Lossy Counting* [84]. Despite its approximation nature, its error is guaranteed not to exceed a user-specified parameter ϵ. This algorithm maintains a data structure D, which is a set of entries of the form $(x, \text{freq}(x), \text{err}(x))$, where x is an item set, $\text{freq}(x)$ represents its approximate frequency, and $\text{err}(x)$ is the maximum possible error in the frequency. Initially, the D is empty and the algorithm updates it as follows.

1. Batch: the algorithm divides the incoming transaction stream into buckets. where each bucket consists of transactions $\lceil \frac{1}{\epsilon} \rceil$. The buckets are given incremental bucket IDs starting from 1. We accumulate as many buckets as possible into available main memory and process them in a batch fashion. We let $q_{current}$ denote the bucket ID being processed and let β denote the number of buckets in the current batch.

2. Update Entry: For each element $(x, \text{freq}(x), \text{err}(x))$ in D, first update the $\text{freq}(x)$ by counting the occurrences of x in the current batch. If the update entries satisfies $\text{freq}(x) + \text{err}(x) \leq q_{current}$, we remove this entry from D.

3. Insert Entry: If an item set x satisfies $\text{freq}(x) > \beta$, and the set x does not occur in D. We then insert a new entry $(x, freq(x), q_{current} - \beta)$ to D. Note that not all possible item sets in B need to be examined, because if a set does not satisfy the condition, none of its supersets would do.

This algorithm ensures that every item set x makes its way to D has a true frequency f_r that satisfies $f_r \geq \epsilon N$, where N is denoted as the length of the processed stream. This algorithm also guarantees that $\text{freq}(x) \leq f_r \leq \text{freq}(x) + \text{err}(x)$. When a user requests a list of frequent item sets with a support threshold s, we can output those entries in D, which satisfy $\text{freq}(x) \geq (s - \epsilon)N$.

We conduct a preliminary experiment using the lossy counting algorithm on a Busybox Linux disk image. We observe that a significant portion (approximately 76%) of the discovered frequent itemsets are sequential (i.e., blocks ID inside those sets are sequential). It indicates that the simple sequential prediction may already deliver satisfactory accuracy and can be a fallback solution if we do not have sufficient block access history to apply the lossy counting. We also analyze the non-sequential sets and notice that most of them correspond to the behavior of accessing filesystem metadata blocks (e.g., inodes). These blocks are usually stored far away from actually file data, thus the sequential prediction will not work on them.

3.6.2 Graphics Cards

Conventional hypervisors provide a simple frame buffer abstraction for the graphics hardware. Briefly speaking, the guest OS perceives the graphics hardware as a portion of regular memory. To display an image on the screen, the guest simply copies the corresponding bitmaps to the memory buffer. This approach assumes each bitmap is rendered entirely using software. In practice, software rendering works well for most text-based applications, but can lead to poor performance for graphically intense applications.

In BrowserVM, we present an alternative emulation approach for graphics hardware. It enables the guest OS to interact with the host GPU to accelerate graphics rendering directly. To better demonstrate our implementation, we briefly introduce how GPU hardware works. Modern GPUs typically disallow an application to manipulate their low-level registers and memory space directly. Instead, an application can only program the graphic pipeline through specific high-level libraries provided by the GPU vendors. Commonly used libraries include DirectX, Vulcan, and OpenGL. They typically feature a client-server model, as shown in Figure 3.3. An application written to use those graphics APIs is the client and runs on the CPU. Upon invoking an API, the client packs the parameters into a command buffer, which is then transmitted to the server. The server, typically running on the GPU, will interpret the received commands and performs the actual graphic computation. Our approach's main

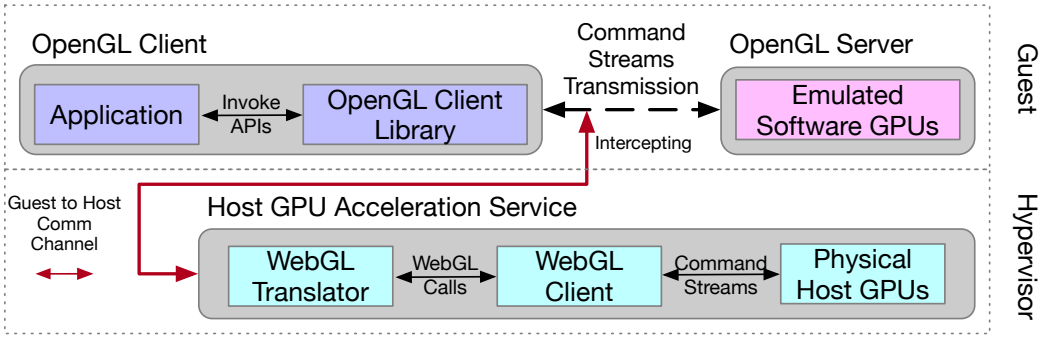


Figure 3.3: BrowserVM enables guest applications to use WebGL to accelerate 3D rendering.

idea is to intercept the command streams issued by a guest application and redirect them to the host GPUs for execution.

Although this idea seems simple, implementing it entails a technical challenge. We do not have access to the source codes of existing multimedia applications and graphics libraries. We thus cannot directly modify them to enable graphics command interception and redirection. To address this issue, we need to adopt a technique named hooking. Specifically, hooking is the process of intercepting a binary program's execution at a specific point, typically entries of functions, in order to alter or augment its behavior. A common implementation of hooking is dynamic linker hooking; we can instruct the dynamic linker to inject user-provided codes to override the hooked functions during the startup of an application.

We demonstrate how to use this approach to deal with OpenGL (the most widely used graphics library in Unix-based OSs) and the core principles used here can be easily applied to other libraries. Specifically, an OpenGL-based application can invoke OpenGL graphics APIs in three different ways:

1. An application directly compiles with an OpenGL library so that it can directly call the OpenGL APIs.

2. An application utilizes the eglGetProcAddress function to get pointers to the OpenGL APIs at runtime.

3. Less likely, an application uses system calls dlopen and dlsym to dynamically load the OpenGL library at runtime.

For the first case, we implement a dynamic library containing wrapper functions for all the OpenGL APIs. The wrapper functions contain the necessary logic for graphics command interception and redirection. We then use the dynamic hooking technique to inject the wrapper library to replace the original OpenGL APIs. In a Unix-based system, this can be easily achieved by setting the LD_PRELOAD environment variable to the wrapper library's path. Regarding the second case, we again use dynamic hooking to rewrite the eglGetProcAddress function such that it directly returns the pointers to our wrapper functions. Similarly, we handle the third case by overwriting the dlopen and dlsym functions so that they load our wrapper library in preference of the original OpenGL library.

It should be noted that upon arrival of the command streams in the host side, we need to adapt a translation library [34] to convert the OpenGL commands to WebGL commands before execution. WebGL is the only JavaScript API that a browser application can use to interact with underlying GPU.

3.6.3 Network Adapters

BrowserVM aims to emulate an Ethernet adapter to support networking applications. The emulation is challenging because browsers disallow applications from accessing raw network sockets. The closest substitute we can find is WebSockets, which allow applications to make outgoing full-duplex TCP connections with WebSocket servers. However, WebSockets are insufficient for emulating a hardware Ethernet adapter due to two reasons. Firstly, WebSockets cannot accept incoming connections, which means we cannot support server-side applications. Secondly, newly-opened WebSockets perform a specifically-designed handshake that promotes an outgoing HTTP connection to to a WebSocket connection. Existing socket-based servers and clients, however, expect a standard TCP handshake. As a result, they will not be able to send or receive WebSocket connections out of the box.

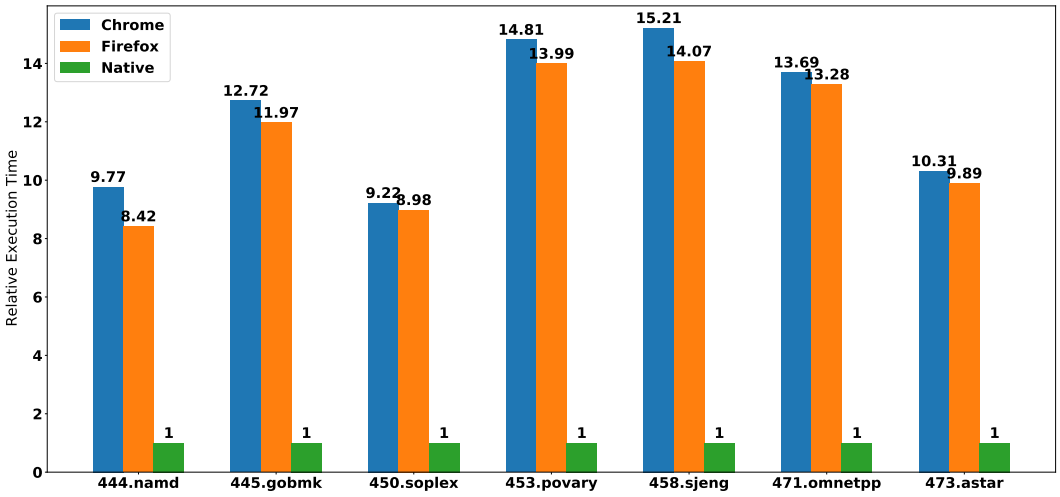


Figure 3.4: Relative Execution Time of Seven SPEC CPU Benchmarks

To circumvent the above problems, BrowserVM utilizes a technique called Ethernet tunneling. It allows us to encapsulate layer 2 Ethernet frames within a higher-level packet format native to the transit network, in our context, WebSockets. The tunneling relies on an in-kernel virtual network interface called TAP. A TAP device looks like a regular Ethernet adapter to applications. Behind the scene, Ethernet frames sent via the TAP device are not going out through a physical wire. Instead, they are delivered to a user-space program attaching to the TAP device. The program then can encapsulate the incoming frames in WebSocket packets and transmit them (using a special interface provided by the hypervisor) to a dedicated WebSocket server acting as a virtual switch. Upon message reception from the switch, the program then can extract the Ethernet frames and inject them via the TAP device to the kernel network stack. To the applications, it would look like the TAP device is receiving data from a physical wire. In this setting, all virtual machines are located in the same virtual private network (VPN) and can communicate using a private IP.

3.7 System Evaluation

In this section, we provide a performance evaluation on BroswerVM's CPU emulator using an established CPU benchmark suite *SPEC CPU 2006* [54].

Performance comparison with a native virtual machine. We conduct the evaluation in a computer equipped with a 6-core 3.8 GHz AMD 3600x processor and 32GB of 3200 MHz DDR4 RAM. The benchmarks are first carried out in a native virtual machine hypervisor (QEMU), whose results can serve as a baseline. After that, the same benchmarks are conducted in BrowserVM running in two state-of-the-art browsers, Chrome 81.0 and Firefox 72.0. Both QEMU and BrowserVM are emulating a single-core X86 PC with 1GB RAM. Currently, BrowserVM cannot support more than 1GB memory because most browsers limit the memory allocation size of each WebAssembly module to 1 GB. This limitation is likely to be lifted off in the upcoming 64-bit version of WebAssembly. We configure the emulated PC to boot from a virtual disk image, which contains an embedded Linux system *Busybox* and seven benchmark executables. These executables are selected because they have a modest memory footprint to avoid using swap space. We also preload their input files to main memory to eliminate the potential influence of disk I/O. Each benchmark is executed five times and the average execution time is measured. For BrowserVM, the translation cache reuse feature is enabled in order to achieve maximum emulation performance. We report the relative execution time in Figure 3.4.

As could be expected, BrowserVM performs worse than native for all benchmarks. On average, BrowserVM is 12.24× and 11.52× slower than native in Chrome and Firefox, respectively. The slowdown may be attributed to two main factors. Firstly, it is widely acknowledged that indirect branch handling is the single biggest source of overhead for DBT [56]. Indirect branching target is only known at execution time and can vary from one execution to the next. DBT has to insert a considerable amount of helper instructions for target identification, imposing a severe runtime penalty. Secondly, WebAssembly has a performance gap with native code. As reported by the previous benchmark [64], WebAssembly bytecode runs slower by an average of 45%. The degradation is mainly due to missing optimizations in WebAssembly compilers shipped

with browsers. Currently, Firefox and Chrome implement their proprietary WebAssembly compilers. As indicated by our benchmark results, Firefox's compiler may be more optimized.

Performance gain from optimization methods used in BrowserVM. BrowserVM's DBT incorporates three optimization techniques, including (1) JIT compilation, (2) generational block management, and (3) translation cache reusing. Note that the optimization (2) is dependent on (1), while the optimization (3) is dependent on (2) and (1). In the following section, we investigate how much performance improvement they can bring about. We conduct performance benchmarks on BrowserVM in five different settings:

(a) Interpretation: all blocks are interpreted.

(b) Conventional DBT: all blocks are compiled.

(c) Enable JIT: only hot blocks are compiled.

(d) Enable generational block management.

(e) Enable translation cache reusing.

We choose four SPEC CPU benchmark executables from the previous experiment. We then run each executable five times and measure the average execution time in Firefox. Note that SPEC provides two different input data sets, namely *test* and *reference*. The reference data set is massive, and it takes several tens of minutes to digest even in a native machine. This size makes it hard to measure the execution time in the setting (a), because interpretation is about two orders of magnitude slower than native and it would take days to finish the benchmarks. Therefore, we decide to use the test data set, whose size is only 1% or 2% of the reference data set. Special care is also required in the setting (b), because our system may encounter several hundred thousand blocks. Compiling them would require several GBs of RAM and Firefox would simply refuse to allocate so much memory and shutdown the tab. To deal with this issue, we need to purge some compiled blocks once the memory pressure is above a certain level. This in theory can be done using a Least Recently Used

(LRU) cache policy. For simplicity, we just remove all the compiled blocks if low on memory. We report the relative execution time in Figure 3.5.

It can be seen that interpretation achieves the worst performance. It is on average 10.9× slower than the most optimal setting (e), where cache reuse is enabled. Note that the setting (e) is used in our previous experiment and the results indicate that it is 11.5× than native. We thus can deduct that interpretation is approximately 110× slower than native. The finding is not surprising because interpretation uses an inefficient read-decode-dispatch loop as discussed in Section 3.2. In the setting (b), we measure the performance of conventional DBT in browsers. Though faster than interpretation, it is approximately 5× slower than the settings (c), (d), and (e), where JIT is enabled. The poor performance is mainly attributed to two factors. Firstly, it pays heavy expenses for compiling blocks that are only executed for a small number of times. Secondly, compiled blocks may be purged from time to time due to memory pressure and recompilation is then required. Another observation is that the setting (c) and (d) achieve similar levels of performance. When generational block management is enabled, approximately 5% performance penalty is paid to merge and recompile old blocks as discussed in Section 3.4. But in return, hundreds of MBs of memory is released. The setting (e) delivers the optimal performance as it can reuse most emulation effort from previous runs.

Use cases of BrowserVM. BrowserVM can be useful for many scenarios that are not compute-intensive. We are recently exploring the potential of using BrowserVM in computer science education. One common study roadblock for inexperienced computer science students is to set up a development environment in their computers. The setup could be challenging due to various reasons, such as operating system differences (e.g., Windows v.s. Mac) and software version conflicts (e.g., Python 2 v.s. Python 3). BrowserVM helps the early learners bypass these issues by instantly creating and playing with a virtual machine with a prepared development environment in a browser. BrowserVM can also be used for software evaluation. Imagine a programmer just develops new software, he can just embed it in a web page and distribute the link to potential new users for evaluation. This can address users' security concerns as they are not

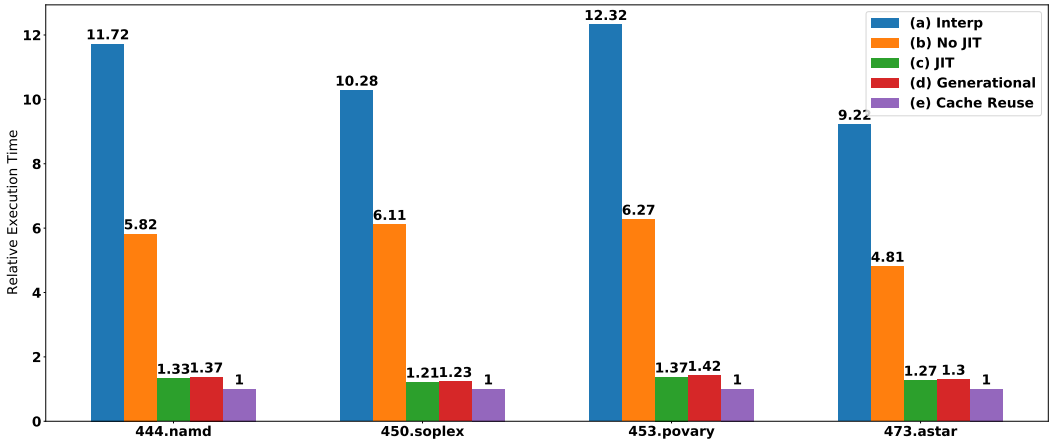


Figure 3.5: Relative execution time for different optimization techniques.

required to download any binary to their computers.

3.8 Chapter Summary

In this chapter, we present BrowserVM, a WebAssembly-based virtual machine hypervisor for browsers. BrowserVM efficiently conducts CPU emulation through dynamic binary translation. It also provides performant hardware emulation for commonly used-hardware, including hard disks, graphics cards, and network adapters. BrowserVM empowers users to conduct full system emulation in browsers, where unmodified and complete operating systems and applications can be executed. We conduct a performance evaluation on BrowserVM and the results show that it is feasible to execute existing non-compute-intensive applications in browsers directly.

Chapter 4

Wasmachine: Bring the Edge up to Speed with A WebAssembly OS

In the previous chapter, we demonstrate BrowserVM, a WebAssembly-based virtual machine hypervisor that enables developers to bundle any unmodified programs in cross-platform applications. It provides a solution to the interoperability challenge in cross-platform development. In this chapter, we shift our focus on the remaining challenge: performance.

A key challenge of WebAssembly is the performance gap with native code: applications compiled to WebAssembly run slower by an average of 45% as reported by previous benchmarks. The slowdown is not desirable, especially for devices with constrained resources such as IoT or Edging computing nodes. The main causes of the slowdown are two-folded. Firstly, conventional WebAssembly runtimes translate WebAssembly instructions to native machine code using just-in-time compilers, which do not apply complex code optimization. Secondly, system calls from WebAssembly applications have to be proxy-ed by runtimes to reach operating systems, which incurs significant performance overhead. To address these issues, we present Wasmachine, an OS aiming to efficiently and securely execute WebAssembly applications in IoT and edge devices with constrained resources. Wasmachine achieves efficient execution by compiling WebAssembly ahead of time to native binary and executing it in kernel mode

for zero-cost system calls. Even when executing in kernel space, Wasmachine stays secure by exploiting various language features of WebAssembly to deliver software-based fault isolation. We implement the Wasmachine prototype in the memory-safe programming language Rust and conduct a performance evaluation. Our results show that WebAssembly applications running in Wasmachine are up to 21% faster than their native counterparts in Linux.

4.1 Motivation

WebAssembly is a young binary instruction format first released in 2017 by mainstream browser vendors [50]. Its primary design goal was to provide an approach to run portable executables with near-native performance in web browser environments. Developers can write their applications in various kinds of high-level languages (e.g., C++ or Rust) and use corresponding compiler toolchains to generate WebAssembly binaries. Nowadays, there has been an emergence of web applications built on WebAssembly for cryptocurrency [70], computer vision [121], and games [144].

More recently, researchers start taking WebAssembly beyond the web. Two research works [63, 51] explore WebAssembly in the context of security and portability for edge computing and IoT devices. WebAssembly has a range of security features such as sandboxed linear memory and structured control flow, which can mitigate many common security vulnerabilities. WebAssembly bytecode is agnostic to source languages and target platforms, enabling a compiled binary to run across different architectures with no additional configuration. Although promising, WebAssembly is not without its flaws. A recent study [64] highlights a drawback for its use in edge/mobile computing: an application compiled to WebAssembly on average runs about 45% slower compared to its native counterpart (i.e., the same application directly compiled to native machine codes). Though the study does not provide any concrete solutions to the problem, it suggests that the slowdown may be due to the implementations of existing WebAssembly runtimes.

A conventional WebAssembly runtime is a program that translates WebAssembly binary instructions to native CPU machine code. The translation is

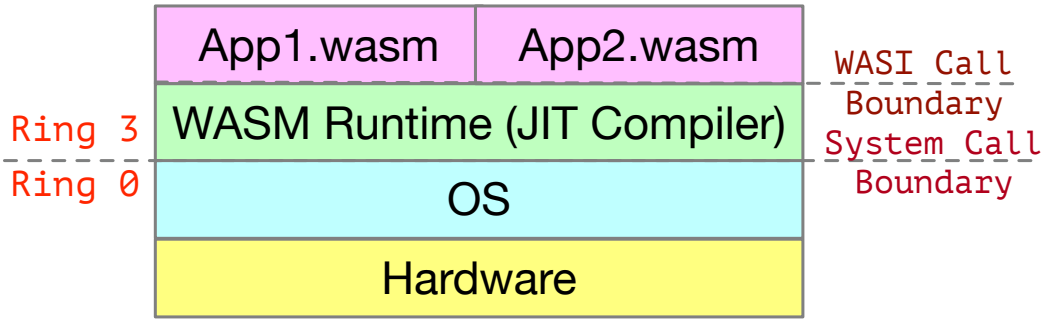


Figure 4.1: Architecture of Conventional WebAssembly Runtimes

most achieved in a just-in-time (JIT) fashion; when a WebAssembly application starts, it will be first interpreted, and after a while, methods frequently executed will be compiled to native codes to improve execution efficiency. JIT enables fast startup time but less code efficiency due to limited time that can be spent on code optimization. For the context of web browsing, the use of JIT is a reasonable choice where fast startup times are important for providing good user experience. However, in the case of IoT and edge computing, code efficiency is preferred.

A runtime can assist a WebAssembly program with system call operations (e.g., networking or file access). Specifically, WebAssembly does not have built-in privileged instructions like system calls and hardware input/output. The only workaround for WebAssembly to execute system calls is to invoke WebAssembly system interfaces (WASI). This invocation process has to go through two boundaries, as shown in Fig 5.1. The first boundary sits between applications and the runtime, where WASI parameters are scrutinized and transformed to match system call prototypes. The second one sits between the runtime and kernel, which triggers context switching from Ring 3 user mode to Ring 0 kernel mode. These boundaries can incur a significant performance overhead, particularly for kernel-intensive applications.

To address these issues, we present Wasmachine, a secure OS aiming to execute WebAssembly applications on bare metal machines in a faster-than-native manner. Figure 4.2 demonstrates the main intuition behind Wasmachine. Given a number of WebAssembly applications, they are first compiled into

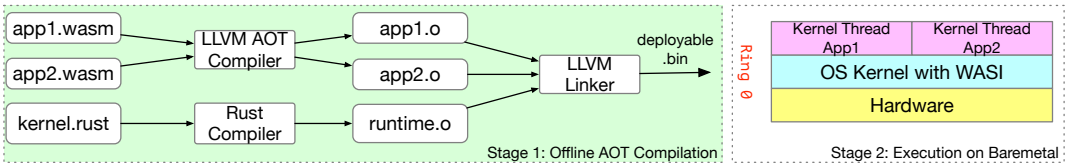


Figure 4.2: Offline Compilation and Architecture of Wasmachine

native binary objects using an Ahead-of-Time (AOT) compiler, which can spend a considerable amount of time on code optimization to generate more efficient binary objects. Resulting objects can then be statically linked (or dynamically loaded) to the Wasmachine runtime. Unlike conventional runtimes that have to run on top of an OS, Wasmachine is an OS kernel for bare-metal machines. The kernel exports system calls in WASI prototypes and runs each WebAssembly application as a kernel thread (i.e., in Ring 0). This design enables WebAssembly applications to invoke system calls as normal functions without incurring additional costs (e.g., parameter normalization and context switching).

Even when executing in kernel space, Wasmachine stays highly secure by the following approaches. First, by exploiting various security features of WebAssembly, our AOT compiler can generate AOT-ed binaries with software-based fault isolation at low runtime cost. Secondly, our kernel provides additional process isolation by a specifically designed virtual memory mapping scheme. The scheme only requires our kernel to navigate a single page table, incurring neither context switch penalty nor page cache invalidation. Lastly, We implement our kernel in Rust, a strong-type system-level programming language that delivers memory safety with zero-cost abstractions.

We consolidated the above techniques and implemented a prototype of Wasmachine on commonly-used hardware architectures x86-64 and aarch64. Our prototype has a low memory footprint, which facilitates deployment to embedded devices with limited resources. We benchmark our prototype on commonly-used kernel-intensive applications, and we show experimentally that WebAssembly applications running in Wasmachine are up to 21% faster than their native counterparts in Linux. We also compared our Rust-based kernel

with C-based kernels and our results suggest that Rust can provide system safety at a low performance cost.

In summary, the main contributions of this chapter are as follows.

1. We propose a secure runtime architecture for WebAssembly, which enables faster than native execution for kernel-intensive applications.

2. We implement an AOT compiler for WebAssembly, which can produce efficient binaries with software-based fault isolation.

3. We implement a WebAssembly-optimized OS kernel in Rust, which provides zero-cost system calls and process isolation.

4. We conduct a quantitative performance evaluation on our Wasmachine prototype. We also conduct a comparison between our Rust-kernel and popular C based kernels.

The remainder of this chapter is structured as follows: Section. 4.2 describes the background and related work on WebAssembly. Section. 4.3 explains the design of our AOT compiler for WebAssembly. Section. 4.4 demonstrates the implementation of a WebAssembly-optimized kernel. Section. 4.5 offers a performance evaluation of Wasmachine and is followed by section. 4.6 concluding this chapter.

4.2 Background

To facilitate readers' understanding of Wasmachine, we provide a background on WebAssembly and its supporting technologies in this section. This section can also be viewed as motivations for using WebAssembly in IoT and edge computing platforms.

Compile Code to WebAssembly and Run Everywhere. WebAssembly bytecode is fully agnostic to hardware architectures and operating systems. It means that a WebAssembly program can be compiled once and run across different hardware architectures without any additional reconfiguration. Although this portability feature is not uncommon in some high-level languages (e.g., Java

or Python), there exists one key difference: WebAssembly was not meant to serve as a compilation target for one particular language. Instead, it is intended to be as open as possible. Nowadays, a variety of compiler toolchains targeting WebAssembly [143][115][109] exist. The most famous example is Emscripten [143], which is the preferred method to generate WebAssembly binary from C/C++. It has empowered many WebAssembly-based browser applications such as cryptocurrency [70], computer vision [121], and games [144]. Emscripten produces not only WebAssembly binaries, but also the associated JavaScript libraries containing runtime support for the WebAssembly code. Specifically, these Javascript libraries are an emulation layer of the C standard library (Libc), providing POSIX system interfaces such as file or socket APIs. It allows developers to port existing code intended to be compiled in POSIX-based systems to browsers with tiny modifications. Nevertheless, the emulation layer comes at a significant performance cost and only works on browser environments.

To address these issues, Mozilla announced a new standard: WebAssembly System Interface (WASI). It defines a set of POSIX-like operating system interfaces that every WebAssembly runtime should implement. As a result, a WebAssembly application no longer needs to ship an emulation layer of Libc. Instead, it can directly invoke the system interfaces provided by the runtime. The WASI standard also opens the possibility of executing WebAssembly programs outside browsers. Recently, many WASI-compliant WebAssembly runtimes designed for non-browser environments are emerging. For instance, Jacobsson et al., [63] implement a WebAssembly interpreter for resource-constrained IoT devices. Hall et al., [51] enhance Google's V8 engine to implement a serverless computing framework for edge devices. Industrial solutions such as Cloudflare [129], Fastly [124] and Wasmer [134] also propose proprietary WebAssembly runtimes to better fit their cloud computing platforms. Clang [71] compiler is now able to generate WebAssembly binaries with WASI support.

Safe and Deterministic Execution in a Sandboxed Environment. WebAssembly provides isolation for running untrusted code (i.e., sandboxing). It is achieved with the help of two security features. First, WebAssembly applies a linear memory model, where an application's memory is laid out linearly and fixed at compile time. It can prevent many well-known security vulnerabilities

and errors arising from direct memory access. Secondly, WebAssembly features structured control flow, which mitigates erroneous control instructions (e.g., jmp and goto) from performing unsafe branching to other parts of the program. It ensures the execution of a WebAssembly program to be deterministic. WebAssembly runtimes can also provide an additional sandboxing layer by deciding to what extent a WebAssembly program can interact with the underlying OSs. For instance, whether a program can access local filesystem or consoles. These features enable us to run a WebAssembly application in kernel space without compromising system security.

A Promise of Near-Native Speed. The WebAssembly specification promises an execution speed as fast as native. As a result of this, we have witnessed many websites rewrite their performance-demanding Javascript code to WebAssembly. According to an empirical study [109], this can obtain up to a 4X performance improvement. However, many benchmarks today still observe that native code executes much faster than WebAssembly. One potential cause is that existing runtimes translate WebAssembly instructions to native binary in a JIT fashion. On the one hand, JIT enables a fast startup of applications, which is vital for web browsing user experience. On the other hand, JIT tends to generate sub-optimal binaries due to the lack of sufficient code optimizations. This argument is supported by a study [64], which examined JIT-ed code and showed that JIT might lead to code bloat, poor register allocations, and unnecessary branching operations. We argue that JIT is suboptimal for IoT and edge computing platforms. Their commonly-used applications (e.g., databases and web servers) are long-running. Therefore a relatively longer startup time is acceptable, and code efficiency is more valuable.

Another aspect that prevents WebAssembly from reaching native speed is the architecture of existing runtimes. As we demonstrated in Fig. 5.1, existing runtimes are running as a userspace program sitting between WebAssembly applications and the underlying OS. With this architecture, each system call from the WebAssembly applications needs to go through two boundaries: applications to runtimes (parameter normalization) and runtimes to kernel (context switching). These boundaries can incur significant performance penalties, especially for kernel-intensive applications.

To overcome these issues, we present a novel WebAssembly OS/runtime called Wasmachine. One contribution of Wasmachine is that it implements an AOT compiler to translate WebAssembly binaries to native machine code. AOT compilation brings about two significant advantages. First, it can generate highly efficient binaries by performing complex code optimizations, which will be considered too costly in most cases of JIT compilation. Secondly, AOT compilation can be done offline (e.g., in a powerful server) to reduce target devices' battery usage. Once the compilation is finished, we only need to ship the AOT-ed binaries (i.e., no need for the bulky compiler) to reduce disk space requirement. It is beneficial for resource-constrained IoT devices.

One key contribution of our compiler is that it can produce binaries with software-based fault isolation at a low runtime cost. Unlike existing WebAssembly compilers that insert costly runtime security checks (e.g., out-of-memory access check) to output, our compiler generates few instructions and offloads most of the checks into our specifically-designed kernel. The other key insight of Wasmachine lies in its Rust-based kernel, which executes AOT-ed binaries in Ring 0. Running applications in kernel space is not necessarily risky and has been tried before with varying success. Spin OS [15] makes itself extensible by allowing users to download and execute modules written in Modula [23] in kernel space. Singularity [60], written in Sing# (a variant of C#), provides a software isolated process (SIP) abstraction and runs its kernel and applications in the same hardware security ring. Linux implements an in-kernel virtual machine for the eBPF bytecode [89] to execute user-supplied networking programs. These works have one thing in common: they maintain system security not by hardware protection, instead by writing the kernel and applications in a memory-safe programming language/bytecode. Considering WebAssembly and Rust are also memory-safe, Wasmachine should maintain the same security level as the above systems do.

To our knowledge, our research is the first to systematically design and document an AOT-based WebAssembly runtime for bare-metal machines and to conduct quantitative performance evaluation. As a viable alternative to conventional WebAssembly runtime, Wasmachine meets the following design goals.

1. **Near-native or faster than native.** Wasmachine fulfills the promise of near-native execution in WebAssembly specifications. Furthermore, Wasmachine outperforms conventional WebAssembly runtimes when executing kernel-intensive applications.

2. **Process isolation**. Wasmachine provides process isolation to protect each process from other processes on the same machine. The isolation includes memory segmentation (such that an application cannot influence another application's memory or execution) and filesystem segmentation (such that an application may only read and write its own files).

3. **Lightweight.** Wasmachine is sufficiently lightweight (in terms of kernel code size and memory usage) such that it can be installed in resource-constrained embedded devices.

4.3 AOT Compilation for WebAssembly

Our AOT compiler is constructed with the LLVM compiler infrastructure [71]. In detail, we first implement a lexer and a parser for WebAssembly bytecode via Flex and Bison [79]. These tools allow us to iterate each WebAssembly instruction and convert it into LLVM intermediate representation (IR). The conversion is generally not difficult since both WebAssembly and LLVM IR are low-level instructions, and many of them are semantically similar. We provide several simplified examples in Table 4.1. The only tricky issue is that WebAssembly instructions are stack-based (i.e., operands are stored in a stack) while LLVM IRs are register-based (i.e., operands are stored in registers). To facilitate the conversion, we need to maintain an operand stack to know where the operands of each WebAssembly instruction are currently stored. It is worth noting that the operand stack only exists during compilation and incurs no runtime cost because the location of each instruction's operands can be statically determined due to WebAssembly's structured control flow and validation rules [50].

The conversion result may be suboptimal and can be significantly improved with LLVM optimization passes (e.g., constant elimination and loop unrolling).

Following this step, the optimized IRs can be sent to LLVM backends to generate native objects for different CPU architectures. These native objects will be statically linked (or dynamically loaded) to the Wasmachine kernel and operate as a kernel thread. There is a consensus that running untrusted applications in kernel space is risky. Nevertheless, by exploiting various security features of WebAssembly, our AOT compiler can generate trusted binary objects with software-based fault isolation at a near-zero runtime cost.

Memory Isolation. Our compiler delivers application memory isolation by exploiting WebAssembly's linear memory model. In detail, each WebAssembly application has one specifically-designated default linear memory *mem*. It is a contiguous byte-addressable range of memory spanning from offset 0 and extending to a variable memory size. The memory content can be read and written by memory operation instructions 'load' and 'store' with an offset parameter. A simplified example is 'i32.load 10', which loads a 32-bit integer sitting in *mem[10:13]*. To translate WebAssembly memory operations, our compiler first inserts a program initialization routine, allocating a contiguous memory chunk as the linear memory and records its address in a global variable *base_ptr*. Afterward, the compiler transforms each memory operation's parameter by adding the value of *base_ptr*. For instance, the WebAssembly instruction 'i32.load 10' will be transformed into LLVM IR '%address_temp = 10 + @base_ptr; %result = load i32, i32* %address_temp'. Here the symbol '@' denotes a global variable and '%' represents a local variable. Note that reading global variables directly in LLVM IR is not optimal because in LLVM a global variable is not a candidate for register allocation. In other words, a global variable is always stored in main memory, and reading it costs significantly more CPU cycles than a local variable stored in a register. We apply a workaround to fix this issue; if a function body contains more than one memory operation, we will copy the value of *base_ptr* to a local variable *cached_base_ptr* and use the local variable for memory address transformations.

To provide memory isolation, we can inject bounds-checking instructions on '%address_temp' to ensure that all memory operations are constrained in the allocated linear memory region. If out-of-bound access is detected, we can terminate the application immediately to ensure the system is secure. However,

Table 4.1: AriteConversion from WebAssembly to LLVM Bytecode

Instruction Description	WebAssembly	LLVM Assembly
Integer Add	i32.const opa i32.const opb i32.add	%1 = alloca i32, align 4 %2 = alloca i32, align 4 store i32 opa, i32* %1, align 4 store i32 opb, i32* %2, align 4 %3 = load i32, i32* %1, align 4 %4 = load i32, i32* %2, align 4 %5 = add nsw i32 %3, %4
Integer Subtract	i32.const opa i32.const opb i32.sub	%1 = alloca i32, align 4 %2 = alloca i32, align 4 store i32 opa, i32* %1, align 4 store i32 opb, i32* %2, align 4 %3 = load i32, i32* %1, align 4 %4 = load i32, i32* %2, align 4 %5 = sub nsw i32 %3, %4
Multiply	i32.const opa i32.const opb i32.mul	... %5 = mul nsw i32 %3, %4
Performs bitwise logical AND	i32.const opa i32.const opb i32.and	... %5 = and i32 %3, %4
Performs bitwise logical OR	i32.const opa i32.const opb i32.or	... %5 = or i32 %3, %4
Shifts arithmetic right	i32.const opa i32.const opb i32.shr_s	... %5 = shl i32 %3, %4

this software bound-checking inevitably leads to a performance penalty because we need to insert at least two instructions (compare and branch) on each memory operation. To avoid the penalty, Wasmachine offloads bounds-checking tasks to the kernel, which will be discussed in section. 4.4.

Execution Integrity. Another safety guarantee our compiler can deliver is control-flow integrity. It is a safety mechanism that prevents attackers from arbitrarily controlling program behavior (i.e., making unintended control-flow transitions). Generally, there are three types of external control-flow transitions

that need to be protected including:

1. Direct jumps or function calls.
2. Indirect function calls (i.e., function pointers).
3. Function returns

Note that (1) has been protected by WebAssembly's structured control flow, which represents code as an ordered sequence of basic blocks and scoped control flow constructs (e.g., if-else or loop). Notably, goto (jump) statement is deliberately excluded, and branching instructions must point to valid destinations within the enclosing constructs. As a result, the correctness of an application's control flow can be verified at compile time, and its execution is guaranteed to be deterministic. Our kernel further protects (1) by setting the application's code segment as immutable to prevent code injection.

For (2), our compiler exploits a language feature of WebAssembly: WebAssembly uses an instruction named 'call_indirect' to achieve indirect function calls. The instruction takes not only a function address (in a more accurate term: function index), but also the type of the function being called. This extra function type allows us to implement a signature check, verifying that the expected type of the indirect function call matches the function's actual type. This check is crucial for security because it is undefined in C++ to cast a function pointer to another type and call it that way, which can potentially smash stack in some platforms.

Lastly for (3), our compiler stores local variables in a separate user-addressable stack in linear memory. It prevents attackers from overwriting return address by stack smashing techniques such as buffer overflow. Our compiler can also optionally enable function tail call to reduce the chance of stack overflow and slightly increase execution speed. It is worth noting that tail call is explicitly disabled in current WebAssembly specifications as it may lead to endless loop and freeze user interfaces of web browsers. Nevertheless, this is not a concerning problem for IoT or edge devices.

Privileged Call Protection. Another security feature we can harness is that WebAssembly has no built-in privileged instructions such as hardware

input/output or syscall. It ensures that WebAssembly binary alone cannot express and execute any sensitive instructions even if it runs in kernel space. The only workaround to execute privileged instructions is through external APIs (WASI) explicitly exposed to WebAssembly memory space. These external APIs, however, can be easily scrutinized by our compiler at link time. For instance, our compiler can whitelist socket functions to allow Internet access and blacklist file functions to prevent filesystem access. These settings can be configured on a per-application basis. Note that our kernel implements the WASI APIs and offers additional isolation. We will cover these features in the next section.

4.4 WebAssembly-optimized Kernel Implementation

We implement a lightweight kernel to provide better security and efficiency for running WebAssembly applications in resource-constrained devices. This section demonstrates the implementation of our kernel, focusing on areas in which the integration of WebAssembly affects the design.

Kernel Architecture and Programming Language. Wasmachine features a Unix-like monolithic kernel architecture and currently supports 64-bit x86 and armv8 hardware. In the future, we plan to support more modern CPU architectures such as RISC-V. Our kernel is SMP-aware and can run processes in parallel on multi-core hardware. To enable concurrent access to system calls, Wasmachine implements commonly-used locking primitives like spinlocks, mutexes, and semaphores to guard and synchronize kernel data structures.

Our kernel contains approximately 3700 lines of Rust (excluding device drivers, imported libraries, and network stack) and 100 lines of assembly. The footprint is small (188 KB for x86-64 and 137 KB for aarch64 with compression) and easily deployable into embedded devices with limited resources. We use Assembly in our kernel to initialize hardware (e.g., timer and interrupt controller) during boot time and save/restore registers during a context switch. The reasons we use Rust rather than C are multifold. First, C requires extra care to manage memory safely, even then bugs (e.g., buffer overflow, use-after-free, and data

racing) are still common. Rust can easily prevent these issues by enforcing a set of type-safety and memory-safety rules at compile time. In addition to this, Rust provides many convenient abstractions such as Unicode strings and memory pools, which we can reuse to reduce the implementation workload. Lastly, Rust is also sufficiently low-level and can be optimized for maximum performance at a similar level to C [86].

However, one essential language feature of Rust 'ownership' poses a challenge on the kernel implementation. Specifically, given an object T, Rust developers can only access it in one of the following two ways: (1) Have several immutable references to the object, or (2) Have one mutable reference to the object. By enforcing this rule at compile time, Rust's memory safety can be guaranteed. Despite its usefulness, this rule is inflexible for monolithic kernels where several kernel components may wish to have multiple references to a shared object and also mutate it. In order to circumvent this issue, kernel developers have two possible workarounds. A naive approach is to use the 'unsafe' keyword, which allows developers to freely access or modify a mutable shared variable at the risk of data races or invalid memory access. It is likely that memory vulnerabilities are introduced within 'unsafe' blocks, compromising the memory safety of Rust. Therefore, we decide to choose the second approach, which is to use Rust's shareable mutable containers such as *Cell* or *RefCell*. Cell containers allow multiple references to a mutable object by exposing APIs that copy the data in/out to access it (i.e., avoiding accessing data in place). On the other hand, RefCell containers do not duplicate data but allow developers to claim a temporary, exclusive, and mutable reference to the inner value. It is implemented using 'dynamic borrowing', where Rust keeps track of how many times the data has been borrowed at runtime, and throws an unrecoverable exception if it detects an attempt to borrow a value that is already mutably borrowed. Our kernel implementation frequently uses these two containers, and we will discuss the potential runtime overhead in section. 4.5.3.

Process Management. Wasmachine adopts a simplified process management model. Specifically, It does not make use of the 'protection ring'. Unlike convention OSs that run user applications and kernel code in Ring 3 and Ring 0 separately, Wasmachine runs everything in Ring 0. Wasmachine disables the

hardware protection because our compiler has already enforced the software-based process isolation. This design brings about two significant advantages. First, it removes the context switch overhead of each system call, which is now just a regular function call. Secondly, the kernel and WebAssembly applications share the same virtual memory address space (i.e., using the same paging table in Ring 0). It improves memory cache performance since it avoids page cache invalidation due to page table switching. Wasmachine also treats each WebAssembly process as a thread. This simplification is made based on the fact that WebAssembly does not currently support multi-threading. It reduces our implementation workload for specific system calls such as *fork* and *atomic*. Nevertheless, we plan to enhance this process model once the multi-threading feature becomes available in upcoming versions of WebAssembly.

Memory Management. Our memory subsystem has to provide two system calls (i.e., WebAssembly intrinsics) as required by WebAssembly specifications. The first one is *memsize*, which can be used to query the current size of linear memory. By default, only 64KB (one WebAssembly memory page) is allocated when an application starts. We implement this intrinsic with a data structure for bookkeeping memory allocation of each process. The second one is *memgrow*, which can be used to enlarge linear memory up to 4GB if needed. The enlargement is implemented using a demand paging system call *mmap*, which maps more physical memory into an application's linear memory region.

To maintain the system security and stability, the memory subsystem needs to prevent two kinds of malicious or buggy memory access. The first one is unallocated linear memory access (i.e., out-of-bounds access). Our kernel detects this kind of access with the help of paging hardware in CPUs. Specifically, upon access to a virtual memory address not mapped to a physical memory address, a page fault interrupt would be triggered, and we could then terminate the troublesome process. The second one is access to another application's memory. To provide memory isolation among processes, our kernel exploits the fact that WebAssembly memory instructions take a 32-bit unsigned integer as an offset of the linear memory (i.e., maximum 4GB addressable space). Therefore, by arranging linear memory regions of processes 4GB away from each other (i.e., by manipulating the *base_ptr* introduced in section. 4.3), we can

ensure that an application would not be able to access the memory from other applications. One simple virtual memory arrangement is that: we first divide entire virtual memory space into segments of 4GB and number them. Then, we can assign memory segments with odd numbers for each application. Unlike software bounds-checking, our approach inserts no instructions and incurs little performance overhead.

Filesystem and Network. Wasmachine implements a Virtual File System (VFS) that supports popular low-level filesystem formats such as in-memory, EXT2, and FAT32. Wasmachine also implements commonly-used low-level hard-disk drivers such as SATA and SDIO. To provide filesystem isolation capacity among processes, Wasmachine implements a system call *chroot*, which changes apparent root directories for running processes. It provides WebAssembly applications filesystem access restricted to a specific directory.

Wasmachine also enables networking via an off-the-shelf TCP/IP stack 'Smoltcp' and implements commonly used Ethernet adapter drivers. We further enhance the network stack to provide a network segmentation functionality similar to *Docker* [88]. Specifically, each application has its internal IP address not accessible from external networks by default. Developers have to explicitly instruct our kernel to publish an application's ports to the outside world.

4.5 System Evaluation

In this section, we provide a performance evaluation of Wasmachine. We are interested in three key questions: (A) How efficient are native binaries generated by our AOT compiler? (B) How much performance improvement can our WebAssembly-optimized kernel bring? (C) Are there performance gaps between our Rust-based kernel and other C-based kernels? We use three different experiment setups to answer these questions.

4.5.1 Native vs. AOT vs. JIT

This experiment investigates how much performance boost we can obtain by using our AOT compiler. We benchmark the test cases listed in table. 4.2.

Table 4.2: Test cases for AOT vs. JIT compilation

Case	Description
nbody	Model the orbits of planets using a symplectic-integrator.
isPrime	Tests if the prime number $2^{31} - 1$ is prime
mergeSort	Sorts an array of 10000 double elements using the merge sort algorithm.
nsieve	Counts the prime numbers in the range of 2 to 39999 using the sieve of Eratosthenes.
arrayReverse	Computes the reverse of an array with 10000 elements 999 times.

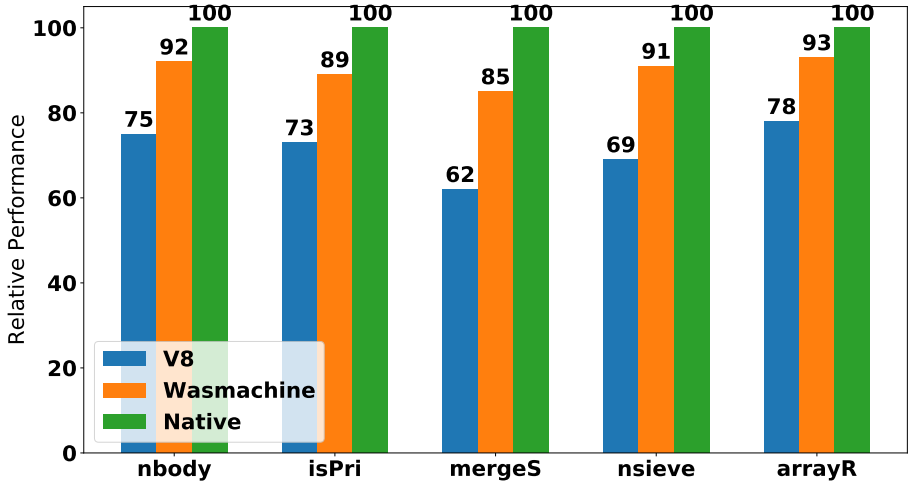


Figure 4.3: JIT V8 vs. AOT Wasmachine vs. Native Performance Comparison

All the test cases are implemented in C++. They are first compiled into WebAssembly binaries using *clang*, an LLVM WebAssembly toolchain. The resulting modules are executed and benchmarked in a conventional JIT runtime *V8* and Wasmachine. The test cases are also compiled into native machine code using a C++ compiler *gcc*, which we use as a baseline. It is worth noting that all the test cases are compute-oriented and do not incur any system calls. This setting allows us to minimize the influence that different runtimes may handle system calls differently. All the tests are conducted on an ARM development board (i.MX 8M) with a quad-core 1.8 GHz GPU and 2 GB RAM.

Figure 4.3 shows the results of the benchmarks. Each bar represents the relative performance compared to native machine code in percentage. A value greater than 100% indicates a speedup, a value smaller than 100% indicates a slowdown. We observe that native binaries can be up to 38% faster than JIT-ed ones. This result matches the observations from the previous study [64]. Another important finding is that AOT receives up to 23% performance boost compared with JIT. This result is expected because our AOT compiler can apply complex code optimization passes that would be considered too time-consuming for JIT compilers. However, we note that at present native code executes at least 7% faster than AOT.

When we further investigate the performance difference between native and AOT and we discovered that the reasons are multi-fold. Firstly, current WebAssembly specifications are missing SIMD (Single Instruction Multiple Data) instructions. As a result, WebAssembly toolchains would not be able to efficiently apply certain code optimizations (especially vectorization) during the WebAssembly generation step. Fortunately, the SIMD extension for WebAssembly has been proposed and is pending for release. We envision that the availability of SIMD will eventually allow numeric-intensive WebAssembly programs to improve their runtime performance and close the performance gap. Secondly, our AOT compiler currently instructs LLVM backends to disable hardware-specific acceleration features (e.g., AVX2) in pursuit of binary portability. As a trade-off, the AOT-ed binaries would not take advantage of target CPUs' acceleration instructions to receive a performance boost. Nevertheless, experienced developers could always specific target CPUs' specific feature pro-

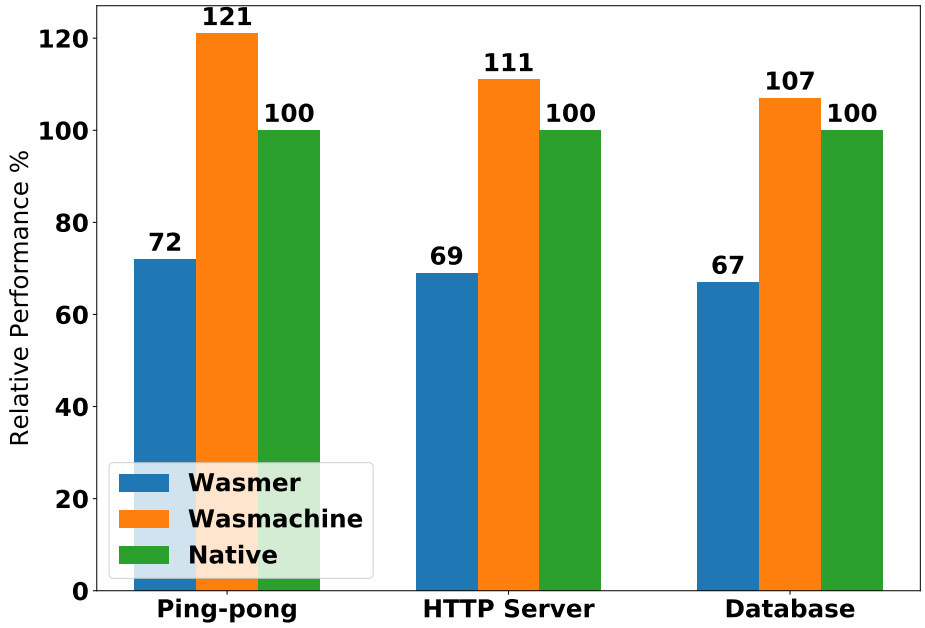


Figure 4.4: Performance Comparison of Kernel-intensive applications

files if portability is not a main concern. Lastly, several security measurements applied by our AOT compiler also incur runtime overhead. For instance, to ensure execution integrity during indirect function calls, our compiler needs to insert signature check instructions in related functions' prologues.

Though our AOT compiler is yet to generate performant binaries in the same league as native ones, it makes contributions: outperforming JIT compilers of conventional runtimes while still harnessing the security benefits of WebAssembly. In the following experiment, we highlight the potential of Wasmachine when the AOT compiler is combined with our specifically designed kernel.

4.5.2 Evaluation of Kernel-intensive Applications

Wasmachine provides a WebAssembly-optimized kernel to reduce system call overhead and to provide security isolation among processes. This experiment

investigates the efficiency of our kernel by benchmarking three kernel-intensive applications as follows.

1. **Ping-pong.** The first benchmark is a 'ping-pong' over a pair of UDP sockets between two WebAssembly processes. Each process takes turns to perform 1500 bytes reads and writes to the other process. Both processes are scheduled to the same CPU so that the kernel will carry out process scheduling between them. The benchmark examines core kernel tasks: system calls, sleep/wakeup, and process scheduling.

2. **HTTP Server.** Libmicrohttpd is a performant and embeddable http server. The server is configured to run in a single-thread epoll mode. Since the epoll API in WASI has a different prototype than the POSIX one, we perform slight modification to the source code. A client runs a stress testing tool called ApacheBench to execute 5000 HTTP GET requests in total and 500 requests at a time. Each request will obtain a dummy 1KB file. After each request, the server will close the TCP connection.

3. **In-memory database.** Redis is an in-memory key-value database. We also modified its source code due to the epoll API prototype mismatch. The benchmark runs one single-threaded Redis server. A client uses a program called redis-benchmark to load over network. The program opens 100 connections to the Redis process and keeps a single GET outstanding on each connection. Each GET requests one of 10,000 keys at random with values of two bytes.

The measurements are conducted on the same machine used in our previous experiment with a dedicated gigabit Ethernet connection. All the experiments are carried out under three settings: WebAssembly in Wasmachine, WebAssembly in conventional runtime Wasmer (as it provides WASI support), and native binaries in Linux 4.19 as a baseline. To stress the CPU efficiency of the kernel, all the benchmarks use an in-memory file system. We present the results in Figure. 4.4. Each bar represents the relative performance compared to native binaries in percentage. As expected, the conventional runtime delivers the worst performance since its JIT compilation already puts it at a disadvantage. One

key observation is that Wasmachine now outperforms its Linux opponent. Ping-pong is running 21% faster in Wasmachine. The HTTP server and database in Wasmachine also surpass Linux by 11% and 7% respectively.

To show that our WebAssembly-optimized kernel is a significant contributory factor to the performance boost, we further measure each application's kernel time ratios (i.e., fraction of time spent in the kernel rather than in userspace). Considering that there is no clear boundary between kernel space and user space in Wasmachine, we redefine the kernel time as the time spent on each WASI call. We present our measurements in Fig. 4.5. One obvious finding is that all the kernel time ratios are above 70%, indicating the tested applications are kernel-intensive. Ping-pong has the highest ratio since it involves little computation and only forwards network packets via kernel calls. In contrast, Redis has the lowest ratio because it carries out a considerable amount of computation such as sorting and indexing in userspace as a database application. What stands out in Fig. 4.5 is the ratio differences among different experiment settings. Applications running in Wasmachine possess significantly lower kernel time ratios than the ones in Linux. In other words, Wasmachine spends much less time on kernel code, which may lead to the performance boost. A likely explanation for the lower kernel time ratio is that all the applications in Wasmachine are running in Ring 0 and sharing the same virtual address page table. Therefore, each system call is just a normal function call, which triggers no expensive context switch and page cache invalidation. It allows CPUs to conserve a significant number of instruction cycles for kernel-intensive applications.

These promising results again justify why it may be beneficial to adopt WebAssembly as a binary format for IoT or edge computing. WebAssembly is already designed with portability and security in mind. With Wasmachine, we can now further harness the near-native or even faster than native execution speed.

4.5.3 Rust vs. C

The most popular programing language for OS kernels is C, due to its high performance, flexible low-level access to memory, and control over memory

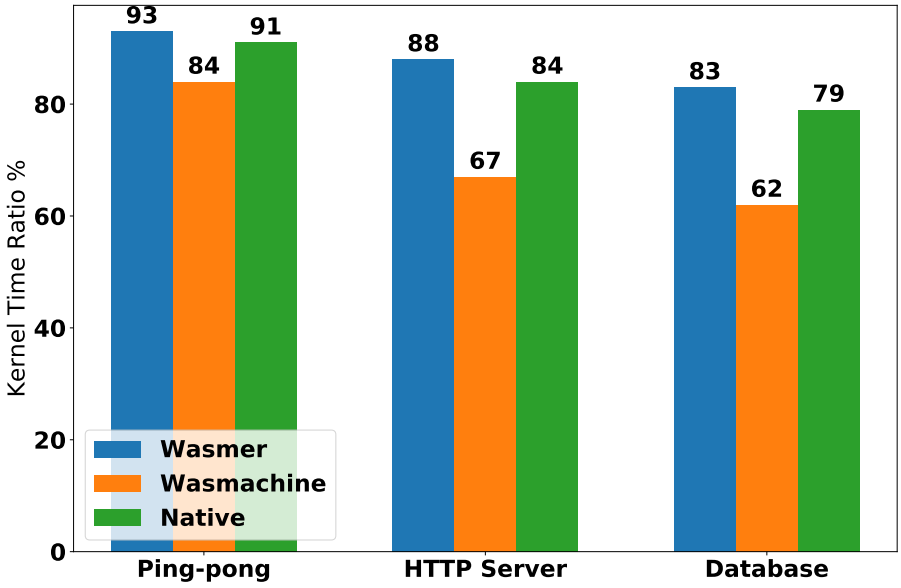


Figure 4.5: Kernel time ratios in different runtimes

management (i.e., allocation and free). However, C-based kernels are commonly susceptible to memory vulnerabilities. Encouraged by recent advances in memory-safe programming languages, we write our kernel in Rust in pursuit of higher system security. Rust, as a sufficiently low-level language, is generally believed to deliver similar performance as C. To justify the choice of Rust, we benchmark the same applications used in the previous experiment on our Rust-based kernel and two other popular C-based kernels: Linux and Littlekernel [81]. To achieve a fair comparison, the tested applications are all running in kernel space.

We realize that the binaries produced by our AOT compiler are specifically-tailored for our Rust kernel. They are not directly compatible with the C-based kernels designed for general purposes. To enable the execution, we make several modifications:

Kernel Preemption. We enable the kernel preemption so that the scheduler is permitted to perform a context switch on kernel code during its execution. It

is crucial for Linux because attempting to run any user code without kernel preemption typically leads to kernel freezing.

Memory Allocation. We modify the kernels to allocate a 1GB linear memory region for the AOT-ed binaries. This memory is sufficient for our experiment. For the sake of simplicity, we do not implement the memory growing and bounds-checking functionalities.

WASI Functions. Another enhancement is to implement WASI functions for the C-based kernels. It is relatively simple, considering that the kernels have already provided a powerful set of system calls as building blocks.

We show our measurement results in Fig. 4.6. Each bar represents the relative performance compared to Linux in percentage. What can be seen is that C-based kernels are up to 12% faster than our Rust-based kernel. A critical factor in the performance degradation may be Rust's ownership mechanism. As we discussed in section. 4.4, Rust disallows multiple kernel components to have a mutable reference to a shared data structure. To circumvent this language limitation, we have to utilize two safe abstractions Cell and RefCell. Unfortunately, they both come with runtime overhead. Cell imposes the cost of memory copies, which may be unacceptable overhead for complex or large kernel data structures. RefCell, despite zero data duplication, has to maintain a reference count and execute the borrow checker before each access. We argue that paying the performance cost for higher security may be worthwhile for systems used in critical scenarios. Furthermore, developers could always eliminate the performance cost by resorting to the 'unsafe' keyword. Research work [80] even suggested that a limited number of unsafe blocks are necessary to form common kernel building blocks. Another likely cause of the performance degradation is that our kernel, as a research prototype, lacks many standard kernel optimizations (e.g., avoiding lock contention, avoiding TLB flushes, and adding I/O caches). However, we believe it could be enhanced by improved implementations.

In summary, our benchmark is not very illuminating about the choice of language, because performance is also affected by differences in the features, design and implementation of kernels. Instead, we could interpret the result positively, which shows us the potential performance of a fully optimized Rust

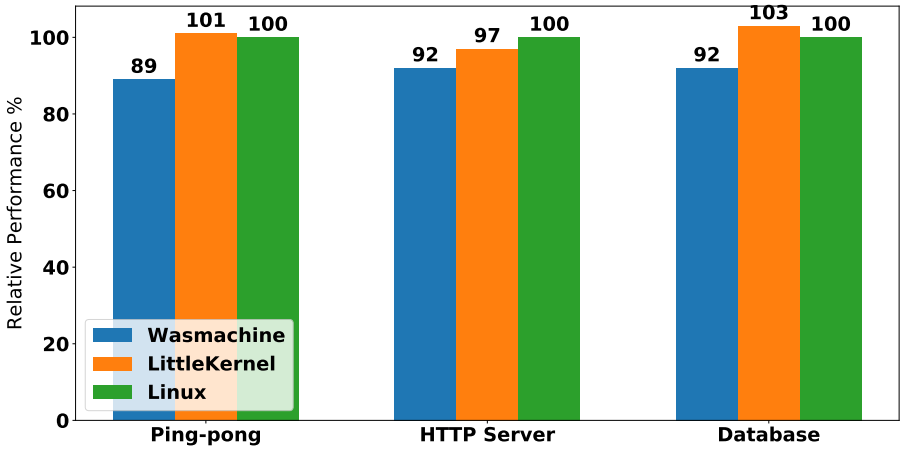


Figure 4.6: Performance Comparison among Rust-based Kernel and C-based Kernels

kernel.

4.6 Chapter Summary

This chapter presents Wasmachine, a secure WebAssembly runtime that can efficiently execute WebAssembly applications in IoT and edge devices with constrained resources. Wasmachine achieves a faster than native execution speed by compiling WebAssembly ahead of time to native binary and executing it in kernel mode for zero-cost system calls. To maintain high security, the compiler and the kernel exploit many sandbox features of WebAssembly to deliver software-based fault isolation among processes. Furthermore, the kernel is implemented in the type-safe programming language Rust in order to ensure memory safety. We benchmark Wasmachine on several commonly-used kernel intensive applications. Our results show that the applications running in Wasmachine are up to 21% faster than their counterparts in Linux.

Chapter 5

Case Study: Exploring WYSIWYG LaTeX Editing on E-Paper

This chapter provides a case study where we use Browserify to build a cross-platform LaTeX document composition system for e-paper devices.

5.1 Motivation

E-paper is a mobile device platform that mimics the appearance of ink on printed paper. A typical application is the use of e-paper as book readers, which has exceeded one billion users according to a survey from Research and Markets 2019 [110]. E-paper has many attributes that make it a favorable choice for scenarios that require high readability. For example, e-paper devices have stable image quality, high contrast, wide viewing angles, and a non-glowing screen. These advantages, especially non-glowing, mean that E-paper is an exciting alternative to conventional glowing LCDs. An increasing number of clinical research works suggest that long-time exposure to glowing LCD screens is likely to induce or exacerbate various physiological problems (e.g., eye fatigue [38], circadian rhythm disorder [127]), psychological issues (attention deficit [32], and increased aggression [67]). These syndromes can affect different user groups in different ways; they may significantly degrade office workers' efficiency and even

pose health risks to underage users. Due to increased awareness of these issues, jurisdictions worldwide have been working on new regulations or laws to achieve screen time reductions or restrictions for workers and children [1]. There is also a so-called 'Going gray' trend where people turn their screens to grayscale to make the glowing screens less stimulating [93]. Non-glowing e-paper gets to the root of the problematic glowing effects and has started drawing attention from researchers.

Most current research works focus on evaluating the readability of e-paper. For instance, Siegenthaler et al. [117] compared reading behavior on e-readers and printed paper. The results suggest that the reading behavior on e-readers is highly similar to the reading behavior on printed paper. Another similar work [116] evaluates the usability of five electronic reading devices and one classic paper book. The results suggested that reading on the LCD-based screens triggers higher visual fatigue with respect to both the E-paper and the paper book, while there exists no significant difference between E-paper and classic paper. There also exists a small number of exemplary applications. For example, Chiu et al. [26] evaluate the potential of using the E-paper powered book readers to encourage students to cultivate healthy reading behaviors. PaperWork [136] presents an approach that allows users to offload their commonly used office applications from a PC to an e-paper device through remote access. Blankenbach et al. [20] proposed a smart medicine package, aiming to address the issue that today's packaging for pharmaceutics provides no information about individual medicine intake. Flexkit [57] and PaperTab [123] explored the possibility of using E-paper as accessory displays for document presentation. However, the academic body of knowledge on the E-paper is still limited. There is little research to explore the potential of e-paper on input-oriented applications, which comprise a central part of both education and office work.

More recently, many manufacturers start producing 13.3-inch e-paper for office and classrooms. They advertise it as a digital device for writing and reading that feels like standard A4 paper. This design philosophy, however, is not entirely implemented. Many devices deliver a decent reading experience to users, but few of them ever attach importance to the writing experience. Their built-in documentation composition applications such as sketching or note-taking

are mostly rudimentary, in a sense that they lack the necessary typesetting capabilities to generate documents with aesthetics and formality. Therefore, they are seldom applicable to scenarios such as education or scientific work. To bridge this gap, we introduce SwiftPad, a document composition system for e-paper devices. SwiftPad incorporates the acclaimed TEX typesetting system, aiming to achieve the following two user experience goals.

High Typographic Quality: SwiftPad delivers excellent typographic quality in the generated documents. They can be used in scenarios that demand formality, such as scientific publications or office documents.

WYSIWYG: SwiftPad provides a WYSIWYG user interface similar to a word processor, allowing users to concentrate on document composition rather than tedious TEX typesetting procedures.

Implementing such a system entails multi-fold challenges. The first challenge is that current e-paper devices lack a standardized operating system (OS). The diversity of the OSs renders a portable implementation of SwiftPad rather tricky. The second challenge stems from the fact that WYSIWYG editing has a strict latency requirement; users expect to see resulting documents in the order of milliseconds after they make some edits. However, existing decade-old TEX engines work in a slow batch processing fashion and may take the order of seconds to compile a long document in e-paper devices with low-spec CPUs. Another related challenge is that as a WYSIWYG editor, SwiftPad has to support dynamic modification of PDF documents generated by LATEX engines. However, a PDF document is essentially a vector graphic, which is generally considered difficult to modify. No reliable and open-source tools on e-paper are available yet for this demanding requirement. The last challenge derives from two notable drawbacks of e-paper screens: low refresh rate and ghost effects. They make e-paper not directly suitable for displaying our WYSIWYG editor, whose contents change frequently.

In this paper, we present practical solutions to cope with the above challenges. Firstly, to make our implementation portable, we build our system on top of our Browserify framework. The generated binary then can be directly executed in browsers of e-paper devices. Secondly, to meet WYSIWYG editing's latency requirement, we propose a checkpointing technique, which allows us to save and

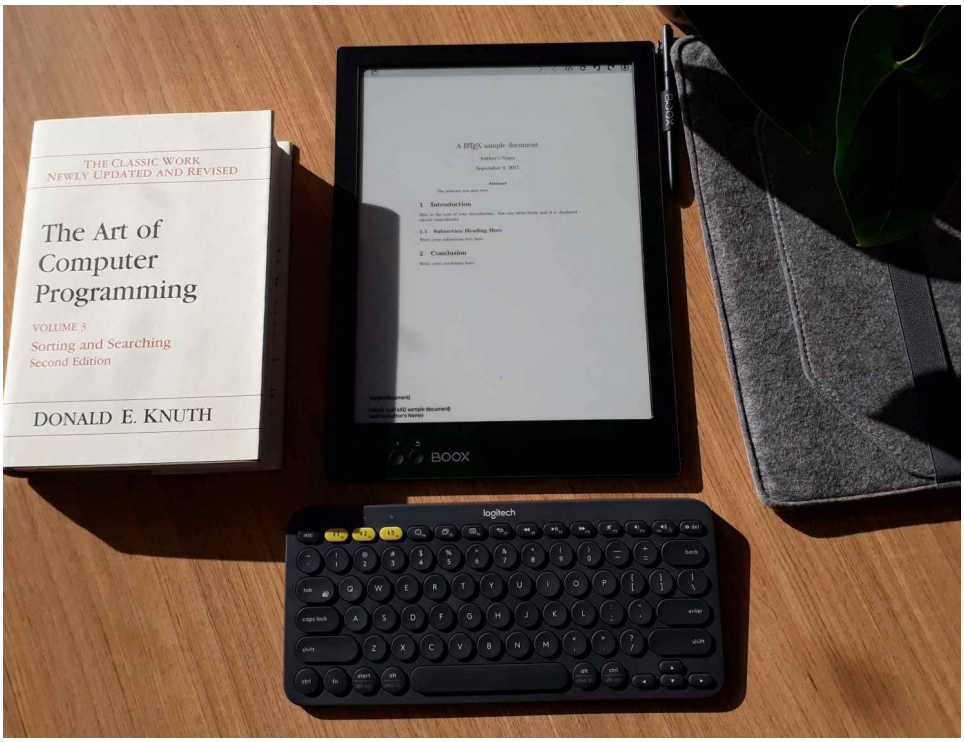


Figure 5.1: General Hardware Setup

restore the LaTeX engine's states. This functionality can be used to accelerate compilation by skipping repeated computation. SwiftPad also proposes an HTML5-based editor, which internally converts vector-based PDFs to semantic-based HTML so that the document contents can be faithfully displayed and easily modified by users. Lastly, SwiftPad exploits the concurrent update feature of e-paper screen controllers to combat the low refresh rate and ghost effects.

5.2 System Overview

Fig 5.1 demonstrates the general hardware setup of SwiftPad, consisting of a 13.3-inch e-paper device and an optional wireless bluetooth keyboard. We argue that keyboards are the most reliable input instrument for SwiftPad because of their popularity and users' familiarity. Nevertheless, we will investigate

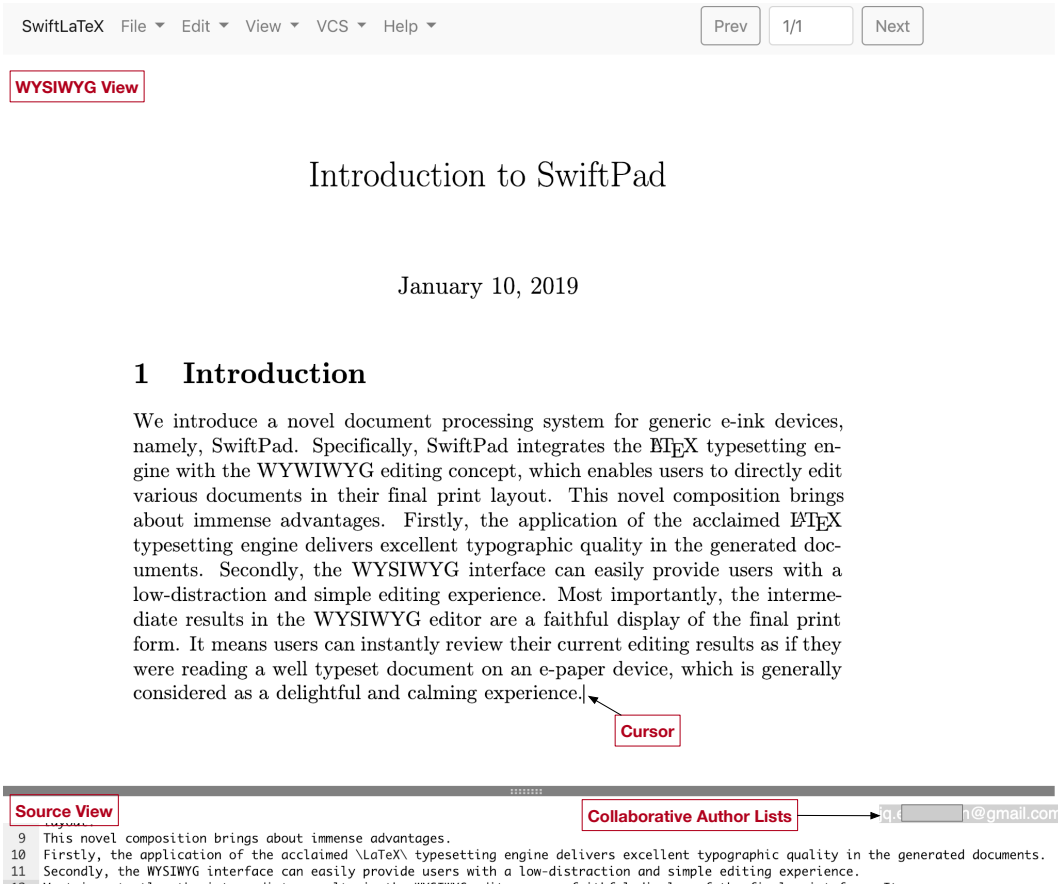


Figure 5.2: User Interface of SwiftPad.

the possibility of other input technologies, such as speech recognition and handwriting input in the near future.

SwiftPad is equipped with a simple-yet-powerful user interface, as shown in Fig. 5.2. Note that the screenshot is directly captured from an e-paper device's graphics memory to pursue better presentation purposes. It can be seen that SwiftPad offers two different editors. The first one is the source editor, which allows advanced users to directly manipulate TeX source code in a classical ASCII editor. The second one is the WYSIWYG editor, which allows the user to directly edit a document in its print form but with an effect on the source. More specifically, the WYSIWYG editor possesses the following features:

1. At editing quiescence (i.e., a moment when the editor has processed all

previous edits of the user, and there are no pending edits that would further change the output), the editor shows the print layout of the PDF document, i.e., acts as a faithful print viewer.

2. At editing quiescence, the user can position the cursor anywhere in the document with the mouse/touchscreen, arrow keys, or a combination thereof.

3. The user can perform edits at the cursor position by typing the keys or backspace and get immediate feedback in the sense that the editor shows a *preview version* of the print view. It is mainly achieved by mimicking the typesetting behavior used in the LaTeX engine. For instance, when a character is being appended, it will automatically inherit the previous characters' font settings to make itself visually agreeable.

4. To retain the modification, the user's editing operations will also be applied at the corresponding position of the LaTeX source code. This functionality requires the editor to possess the ability to infer the source code position of each element in the PDF with character-wise accuracy. We will discuss how to achieve it in the following sections.

5. If the user input pauses, the editor reaches editing quiescence automatically in a reasonable amount of time. It is achieved by replacing the current preview version with the latest compilation result, i.e., faithful output from the LaTeX engine.

SwiftPad also provides a project manager to facilitate users' file management (e.g., uploading or creating source files) as shown in Fig 5.3. The manager incorporates an underlying storage protocol named *RemoteStorage*, which provides offline data access, automatic cloud synchronization and collaborative editing functionalities.

5.3 Comparison with existing systems

The TeX typesetting system by Donald Knuth ushered in an era of high-quality open-source electronic document publishing. Knuth embedded knowledge of

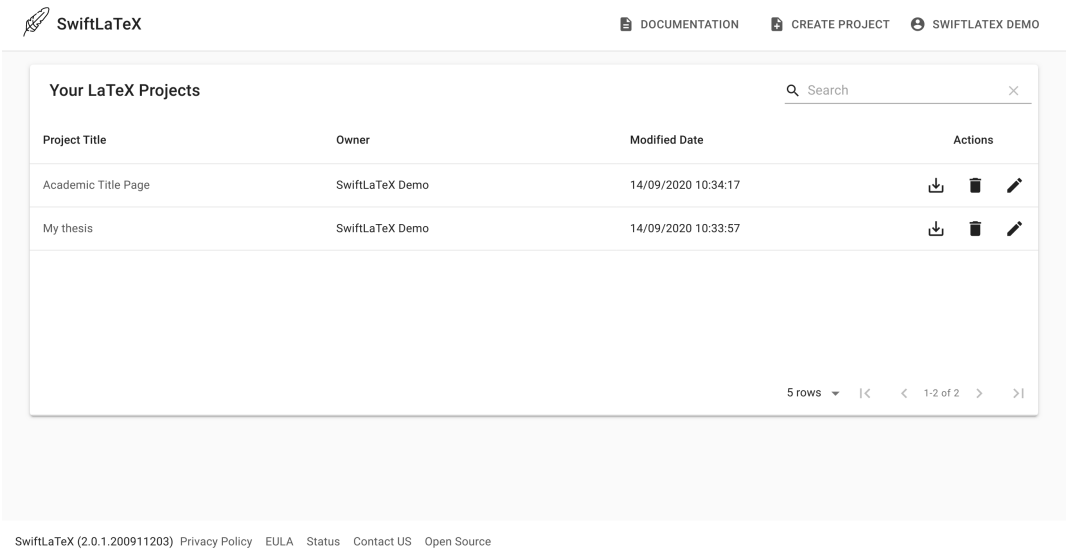


Figure 5.3: Project Management Interface of SwiftPad.

the traditional art and craft of typesetting into TEXÀs a result, TEX produces print-ready documents that adhere to the highest standards of print aesthetics. Over the past decade, various TEX derivatives have been successfully evolved. Among them, LATEX is the most renowned one since it is nowadays used and maintained by a large community and is considered by many to be unsurpassed in quality. Various editing systems have been developed to make LATEX editing more convenient.

The most rudimentary standalone systems are command line based applications (e.g., Latexmk [28] and LatexRun [3]), which commonly serve as automated compilation tools. Given a set of source files, these tools issue the appropriate sequence of commands to communicate with the typesetting engines and generate the resulting document. Generally, the command line programs are rarely invoked directly due to debugging difficulty. Instead, they are commonly utilized as the backend for the graphical editors. Graphical editor applications (e.g., TeXstudio [145], TeXworks, TeXnicCenter and TeXshop [69]) facilitate document editing by providing a graphical user interface (GUI) with various features such as syntax highlighting and automated error correction. These

features significantly reduce the difficulty of debugging and increase editing productivity.

Nevertheless, setting up such an editor may involve complex and laborious configurations (e.g., installing LaTeX packages and maintaining software dependency), which is not user-friendly for novice users. This gap opens opportunities for the emergence of cloud-based editing systems. Compared with the standalone editors, the web-based editing systems possess a distinct advantage, namely zero setup time. Specifically, these web-based editing systems run instantly on at least some mainstream browsers and require no software installation on the users' devices. In other words, web-based systems make LaTeX rapidly and easily accessible in the user's web browsers. Another merit of the cloud-based system is cloud storage, which enables project files to be accessed anywhere globally and considerably facilitates collaboration. These web-based systems generate PDFs on the server-side. Upon the arrival of a compilation request, they have to spawn multiple processes to run LaTeX engines. These processes need to read from and write to the server's file system. If the PDF generation finishes successfully, the generated PDF will be transmitted to the user's browser over the network. If the PDF generation takes too long or the user cancels the generation request, the server needs to send a *SIGTERM* or *SIGKILL* signal to shut down the running processes. If the PDF generation process encounters a fatal error, the server then needs to pipe the output of LaTeX engines back to the client. One major downside of this server-side processing paradigm is that the processing time depends on the overall workload in a server. If many users submit compilation requests to the server simultaneously, they may all have to wait a considerable amount of time to view the generated PDFs. It potentially leads to poor user experiences. To avoid this issue, many service providers, such as Overleaf [97], have to invest a considerable amount of money in deploying more servers. In this work, we adopt a client-side processing paradigm, and all computation is done locally.

Though the above works shed light on SwiftPad, they exhibit several major limitations. Firstly, these products were initially designed for PCs and are not directly applicable to e-paper devices with different hardware specifications and software architectures. Their implementations do not consider the unique

properties of e-paper screens, potentially leading to poor user experience. Instead, SwiftPad presented in this work is easily portable among different mobile platforms and optimized for e-paper devices. Secondly, existing LaTeX systems feature a batch processing workflow; after edit on the source code, users have to wait for the time-consuming compilation process to finish before they can examine whether the resulting document is correct. If not, they have to go back to the editor and pinpoint which line of the source code is causing the problem. Users are now forced to carry out intensive mental mappings back and forth between the essentially one-dimensional textual TeX source codes and the two-dimensional graphics print output [21]. The demanding debugging-like process tends to cause a sense of confusion and frustration. These feelings can be further aggravated on e-paper devices, since it is significantly harder to conduct typing and task switching on e-paper than PCs.

Several partial solutions to this problem have been developed. Among them, cloud-based solutions (e.g., ShareLatex [96] and Overleaf [97]) provide an asynchronous and regular refresh of the print output, which is placed side by side with the code editor. Optionally, the editor also features formatted text inputs, where a fixed number of LaTeX text style commands are displayed in different colors and fonts accordingly (e.g., section header are shown in a large and bold font). Still there is a strong dichotomy between the source code and compiled document and the user has to alternate and switch focus between them in many work cycles. On the contrary, SwiftPad takes the term WYSIWYG literally and allows users to edit faithful print output (i.e., PDF) directly.

5.4 Building Cross-platform SwiftPad

The first major hurdle to overcome is that off-the-shelf e-paper devices lack a standardized OS and programming interfaces. If we adopt a native development approach, we may have to develop and optimize SwiftPad separately for each device. This section walks through the process of using Browserify to build a cross-platform SwiftPad.

The core component of SwiftPad is the TeX typesetting engine. TeX was initially designed to be highly portable and could run on many different computer

systems. Given the same input, TeX should produce identical output (i.e., the same line breaks and page breaks), regardless of the system on which it was running. However, most existing TeX engines were invented decades ago, and the authors failed to foresee the rapid development of modern computing technologies. Many obsolete technologies used in the ancient engines are now undermining the portability. For example, most engines internally adopted a 7-bit character encoding. This encoding scheme is not more supported in modern computers where UTF-8 encoding has become a standard. Most engines also rely on their font rasterization engines to conduct text shaping and render texts into bitmaps. This rasterization approach is not more compatible with modern operating systems, which feature vector-based scalable OpenType fonts. To keep using existing engines in a modern computer, we have to adopt some software compatibility layers. The layers transform some obsolete system calls into modern ones, which vary from one operating system to another. As a result, the TeX document systems have to be adapted and distributed separately for different operating systems. Due to the compatibility layers' implementation differences, these distributions may sometimes even produce different document outputs given the same input. SwiftPad addresses the portability issues with the help of the Browserify framework. Depending on the hardware specifications of e-paper devices, two potential solutions are proposed.

BrowserVM The first solution is to leverage the BrowserVM to run an unmodified Linux operating system along with a completed TeX distribution such as TeX Live or MikTeX in browsers. The process for building PDF then becomes very similar to existing cloud-based LaTeX systems, except SwiftPad performs all computational steps entirely in a browser instead of a remote server. In details, SwiftPad runs GNU Make to read a Makefile, which iteratively executes Xelatex and Bibtex as many times as necessary depending on whether the user has updated input files and whether there is a circular dependency on source files. Xelatex and Bibtex can read required LaTeX packages, fonts, and other resources from BrowserVM's file system, which lazily pulls in files as needed from the network. Xelatex and Bibtex can also write their output to BrowserVM's file system. Once all steps are completed, the Make process exits with a code indicating whether PDF generation is successful or not. If

the Make process exits normally, SwiftPad reads the PDF from BrowserVM's file system and displays it to the user. Otherwise, SwiftPad reads the standard output from Xelatex and Bibtex to the user, describing the error source.

BrowserVM enables the execution of unmodified LaTeX toolchains in a browser at the cost of performance (approximately 10x slower). The slowdown is expected due to the use of dynamic binary translation in BrowserVM. Since the process of building LaTeX documents is typically not compute-intensive, we argue that it may be a viable option to use BrowserVM in specific high-end e-paper devices. For example, a BOOX MAX3 e-reader is equipped with an octa-core 2.0 GHz ARM processor and 4GB RAM. Compiling a single page IEEE document in BrowserVM on the e-paper device costs approximately two seconds.

Wasmachine However, most off-the-shelf e-paper devices possess limited computational capacity (e.g., a single-core 1 GHz processor and 512 MB memory). These low-end devices may not meet the minimum system requirement of BrowserVM. Therefore, we propose the second solution, which is based on Wasmachine and optimized for resource-constrained devices. The first step is to trans-compile TeX engines using Emscripten, a C++ to WebAssembly compiler. The overall build process is familiar to individuals who are used to standard compilation in a Unix-based system. Specifically, TexLive uses the Autotools build system. To initialize the building process, we can invoke *configure* to manually set up compiler-related environment variables. Alternatively, we may invoke *emconfigure*, an automation wrapper script that overrides standard GCC compiler tools with Emscripten alternatives. After the configuration step, we can invoke GNU Make to generate WebAssembly binaries for TeX engines.

However, the WebAssembly-based TeX engines cannot be run in a browser directly because they need filesystem access to LaTeX packages, fonts, and other system files to function properly. Since modern browsers do not have intrinsic filesystem support, we need to provide a filesystem emulation layer for browsers. Emscripten has provided an in-memory filesystem, allowing developers to preload files to memory before a WebAssembly application starts running. However, a complete TeX Live distribution includes approximately 60000 individual files and takes up 16 GB disk space. It is impractical to preload

all of them into memory. Fortunately, an average LaTeX document tends only to reference a small subset of these files, so we only need to preload these frequently used files. We also extend Emscripten's in-memory filesystem with a lazy loading feature, which supports on-demand file downloading from the network upon first access to non-preloaded LaTeX files. The browser caches these files automatically, making subsequent access much faster. To build a PDF, SwiftPad iteratively executes XeLatex and BibTeX until it can guarantee that additional runs will not change the output. This procedure is slightly different from the BrowserVM's solution, where the GNU Make was invoked. We did not apply GNU Make here because GNU Make requires several system calls (e.g., fork and execvp), which are missing in a browser context. Instead, SwiftPad implements its simplified Make system to build a LaTeX Project.

Our preliminary performance measurements demonstrate that the WebAssembly-based engines are approximately two times slower than their native opponents. This slowdown can be alleviated with the help of Wasmachine. Instead of relying on the in-built browser to parse the WebAssembly-based engines, we can feed the WebAssembly-based engines to Wasmachine's AOT compiler to generate native binaries. These native objects will be statically linked (or dynamically loaded) to the Wasmachine kernel and operate as a kernel thread. Using Wasmachine requires us to implement a Wasmachine kernel separately for each e-paper device. Fortunately, most e-paper devices nowadays are built atop Linux so that we can just implement our Wasmachine kernel as a Linux kernel module. We only need to implement the module once since Linux kernels usually maintain module compatibility across different kernel versions.

5.5 Making LaTeX WYSIWYG

We now have a portable LaTeX engine that can run across multiple platforms. The next step is to enhance the engine to provide WYSIWYG functionality. However, implementing the system entails three major challenges. First, allowing direct editing of the PDF document requires that the product of the LaTeX compilation must be combined with the current user edit. However, the combination is considered challenging because the compilation and editing are carried out

asynchronously. Second, the editor is expected to intercept the modification on the PDF document and synchronously apply the alternation in the source code's corresponding position. To achieve this, the editor requires the ability to infer each element's source code position in the PDF in character-level accuracy. It is a demanding target since the highest level of accuracy the existing open-source tools can achieve is merely line-level.

In this work, we present practical solutions to cope with the above challenges. Specifically, to correctly combine the product of the LaTeX compilation and the user's editing, SwiftPad proposes an asynchronous merging mechanism. To infer the source code positions, SwiftPad explores an advanced text-matching algorithm and dynamically patches the LaTeX engine with a bookkeeping mechanism to pursue character-level accuracy.

5.5.1 Asynchronous Merge of Edits

SwiftPad provides a WYSIWYG view, an editable PDF viewer that allows the user to directly edit a document in its print form, but with effect on the source. To start an edit, the user simply clicks on any visible text that should be modified. A text cursor will appear, indicating that the viewer is now ready for the user's input. Afterward, all editing operations such as character appending or removal will instantly be rendered on the viewer, i.e., *preview version*. Meanwhile, to retain the modification, the user's editing operations will also be applied at the corresponding position of the LaTeX source code. After the re-compilation process, the user will automatically be presented with the new *revised version* of PDF.

To provide an authentic experience of WYSIWYG editing, SwiftPad puts considerable effort into generating a preview version that possesses minimal difference from the corresponding revised version, as depicted in Fig 5.4 (b) and Fig 5.4 (c). It is mainly achieved by mimicking the typesetting behavior used in the LaTeX engine.

For instance, when a character is being appended, it will automatically inherit the previous characters' font settings to make itself visually agreeable. Though the typesetting imitation approach is feasible for minor editing, it possesses certain limits when dealing with lengthy edits. Consequently, without

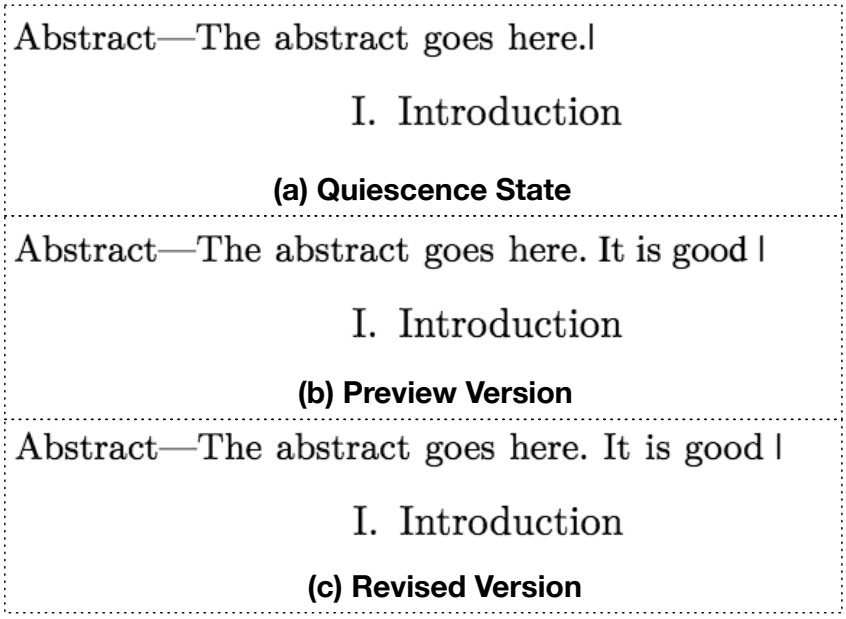


Figure 5.4: Three Editing States in the WYSIWYG View.

any further precautions, the difference between the previewing and revised versions would gradually accumulate along with the user's input.

To address this issue, one idea is to compile the source periodically and base the preview version on the newly-obtained revised version. Therefore, the accumulated difference is cut back and remains unnoticeable most of the time. However, implementing such a mechanism entails a challenge, which is mainly attributed to the nature of asynchronous communication; the user can keep on editing while a new version is already in the compilation process. We depict the predicament in Fig 5.5 (a) using a timing chart. The chart contains two separate timelines. The topmost timeline is indicating the duration the user spends on editing. Likewise, the other timeline is showing the timespan that the background compilation process consumes. It can be observed that at the moment t_1, a background compilation request is initiated. Meanwhile, since SwiftPad is asynchronous, the user continues his editing regardless of the background compilation process's status. At the moment t_2, the compilation is finished, and the revised PDF is available for display. However, it should be noted that this revised PDF is an output of the source code at the moment

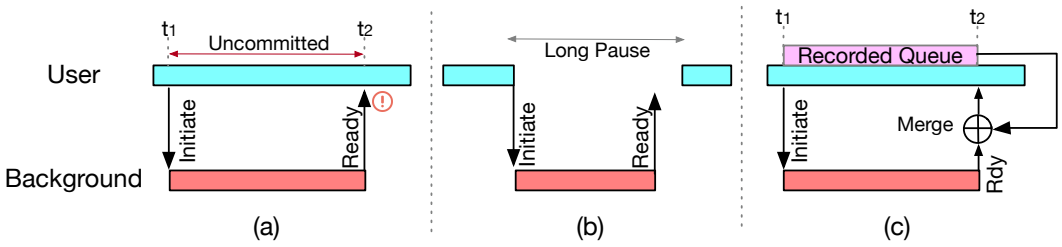


Figure 5.5: Timing Charts in the WYSIWYG View.

of t_1. It does not reflect the user's inputs within the timespan between t_1 to t_2, referred to as the *uncommitted edit*. Consequently, directly displaying the newly-obtained revised version would erase the user's uncommitted edit from the screen. That would be inappropriate because the user's editing process would be severely disrupted.

One naive way to bypass this issue would be to take advantage of the user's pauses in editing, as shown in Fig 5.5 (b). Specifically, it is not uncommon that a user may occasionally take a break from typing. Only if such a break is detected, a background compilation would then be initiated. If the compilation is finished before the user restarts his editing, the resulting PDF can be safely displayed. This approach essentially uses the aforementioned concept of editing quiescence to achieve consecutive safe updates of the PDF that do not interfere with the user input. Nevertheless, this method is prone to starvation (i.e., the long delay of the compilation process) if the user is typing swiftly and continuously for a long time.

Instead, SwiftPad adopts a more sophisticated approach that does not rely on the user's pauses. The intuition is depicted in Fig 5.5 (c); the revised PDF, before being displayed in the viewer, is 'merged' with the uncommitted edit so that the user's editing process will not be disrupted. It can be achieved as follows. First, when the background request is initiated at t_1, our system starts recording all the editing operations in a queue data structure. When the revised PDF is available at t_2, all the operations are dequeued from the queue and replayed on the revised PDF sequentially. This workflow generates a new previewing PDF, simultaneously containing the latest revised PDF and the

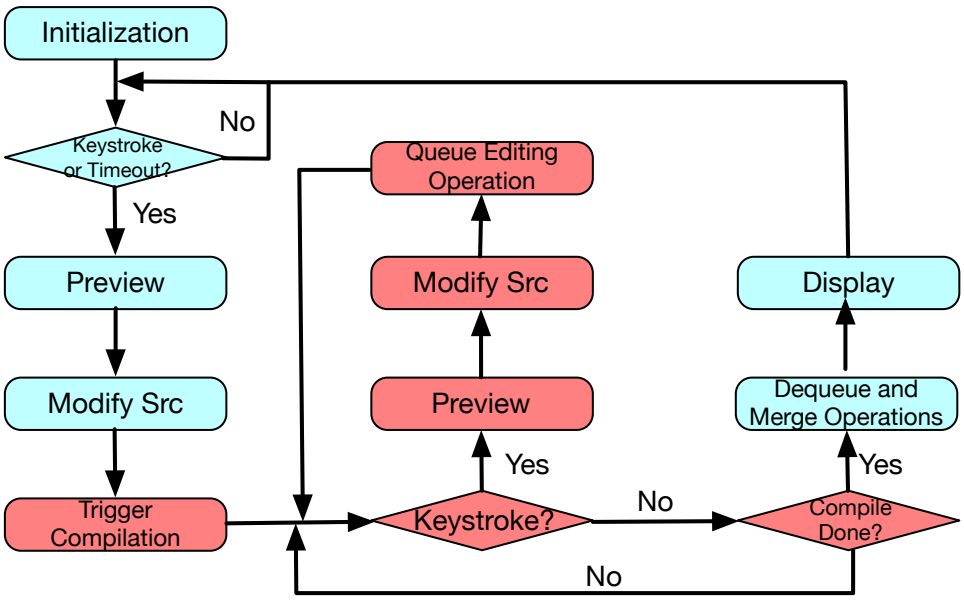


Figure 5.6: Workflow for the WYSIWYG view.

uncommitted edit. Thus, it is safe to be displayed in the viewer. The cursor is repositioned at the correct logical position in the new PDF, and the user can continue editing without interruption.

We summarize the aforementioned design with a flowchart shown in Fig 5.6, where the blocks with an ongoing compilation process are colored in red. And a new compilation process will be triggered whenever either a keystroke or a timeout event (i.e., periodically trigger) is detected. This ensures the users can obtain the latest revised version as soon as possible.

5.5.2 Achieving Fine-grained Source Code Positioning

As a WYSIWYG editor, SwiftPad intercepts the modification on PDF elements (e.g., words or equations) and directly applies the corresponding alternation on the source code. To achieve this, SwiftPad is required to infer the source code position of the modified PDF elements. Such an ability is commonly referred to as LaTeX input-output synchronization, which has been explored by a LaTeX plugin named SyncTeX.

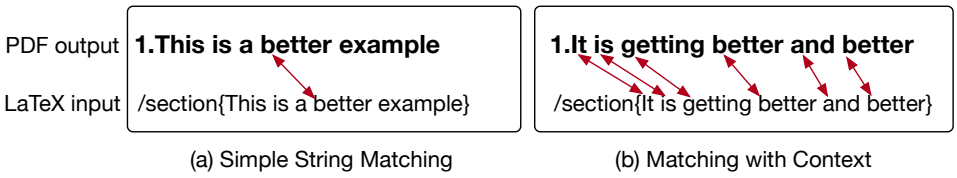


Figure 5.7: Two typical examples for source code position inference.

SyncTeX is designed for conventional typesetting editors, which usually incorporate two views; one for entering LaTeX source code, the other one for viewing the resulting PDF. Generally, the source code and PDF are too long to fit a window, and thus only certain parts of the contents are visible to a user. To facilitate the user's editing process, the two windows are required to be synchronized, such that they are displaying roughly the same part of the document. Solely taking this use case into account, the author of SyncTeX deems it sufficient to provide line-level accuracy, allowing mapping from a PDF element to a whole line in input files [73]. This coarse-grained accuracy is unable to meet the requirement of SwiftPad. To boost the synchronization accuracy from line-level to at least word-level, we have developed and tested two approaches. Firstly, we have developed an advanced text-matching approach. Secondly, we have enhanced the latex engine to achieve character-wise synchronization.

Text-Matching Approach. For a better understanding of the text matching approach, Figure 5.7 depicts two typical cases, each of which contains one line of LaTeX source code and its corresponding PDF output obtained by SyncTeX.

Consider the first case in Fig 5.7 (a) where we wish to infer the source code position of the word 'better' in PDF. A straightforward solution is to run string matching algorithms on the line of source code, which reveals the correct position. However, this approach may cause ambiguity if the word 'better' occurs more than once in a sentence, as shown in Fig 5.7 (b). This exception naturally motivates us to take the surrounding words (i.e., context) of the target into consideration, specifically matching a sequence of words instead of an individual target. Following this logic, we can formulate the source code position inference problem as follows.

Given two character arrays L and P, which represent a line of LaTeX source codes and its corresponding PDF output string, respectively. Besides, L_j denotes the j-th character in the array L and $1 \leq j \leq len(L)$. Likewise, P_k represents the k-th character in the array P and $1 \leq k \leq len(P)$. A matching f between L and P is defined as a function that satisfies

$$P_k = L_{f(k)} \qquad \forall k \text{ in } [1, len(P)]. \tag{5.1}$$

Here we safely assume every character in P has its matching on L. If there exists no matching for a specific character (e.g., the numbering characters in section header), we simply mark it as uneditable and remove it before proceeding.

Meanwhile, we say a matching is valid if it is monotonic, which can be illustrated as follows.

$$f(i) < f(j) \quad \text{iff } 1 \leq i < j \leq len(P). \tag{5.2}$$

This property guarantees the order of the characters in P is also preserved in L.

There may exist multiple matching and we devote the fittest matching f_b as the one which satisfies the following requirement:

$$f_b = \arg\min \sum_{2}^{len(L)} (f(k) - f(k-1)), \tag{5.3}$$

where f represents a valid matching. This requirement contains a simple intuition; for two consecutive characters in P, the distance between their matching characters in L is also minimized under the fittest matching f_b.

It can now be easily seen that the problem's answer happens to be the fittest matching f_b. This mapping can be swiftly found even in a brute-force manner, as the search space is limited in a line of source codes. Moreover, the search space can be further reduced by specific optimizations. For instance, the string matching algorithm can be first utilized on words that only occur once to confirm their matching immediately.

Nevertheless, such an approach is still prone to matching failure. The main culprit derives from the precision of the SyncTeX, which is ± 1 line [73] in most cases. The precision significantly degrades when the element is generated by a LaTeX macro (e.g., math equations or section headers). The incorrect

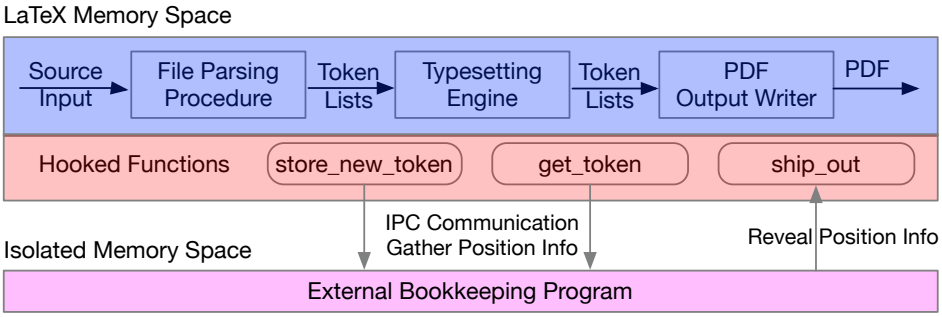


Figure 5.8: Internal Mechanism of the Engine Patch.

line number negatively impacts the reliability of our approach. The unstable technique now forces us to develop a more sophisticated approach, which increases the synchronization accuracy from the internal perspective of LaTeX engines.

Enhancing TeX Engine. Existing open-source LaTeX engines possess no mechanism to track down the source code positions for each PDF element. This work attempts to enhance the engines with a bookkeeping mechanism that provides character-level synchronization accuracy.

The main challenge of implementing such a functionality lies in minimizing our modification's potential interference to the engines. Specifically, the original implementation of the LaTeX engine is generally considered a very complicated code that is difficult to modify safely. Directly patching this source code to alter the internal data structures or execution logic is considered not advisable from a software reliability engineering standpoint, since an under-optimized patch is likely to result in unheeded bugs. Therefore, to avoid undermining the reliability of LaTeX engines, we adopt a neutral observer design in our patch, as depicted in Fig 5.8.

It can be seen that the patch is partitioned into two essential parts. The first one is an external bookkeeping program. It runs in its own memory space and passively receives the position-related information from the LaTeX engines in file parsing and typesetting phrases. The information is obtained by the second part of our patch, which is a specifically-designed hooking library. In short, hooking is the process of intercepting a program's execution at a specific

point, typically entries of functions, to augment the program's behavior. The merit of the hooking technique is its unintrusiveness. Depending on the specific hooking techniques, developers may need neither modification nor re-compilation of the source codes. For example, one commonly-used hooking technique is dynamic hooking, which allows developers to inject extra functions into LaTeX engines' memory space via special instructions to program linkers on the run-time. Moreover, the augment functions in our patch do not alter the memory content of LaTeX process. The neutral observer design ensures the reliability is untarnished.

To better understand how the bookkeeping patch keeps track of the source code positions, it is reasonable to introduce the procedure that LaTeX engines utilize to parse input files. Briefly speaking, LaTeX digests the input files character by character. These characters are generally stored and organized in a data structure named *token lists*, which are essentially linked lists. A token list can represent a character array to be printed or a parameter for a macro call (e.g., 'Paper' in \title{Paper}). Depending on the macros' implementation, a parameter token lists can be moved around in the memory by being duplicated and removed several times before they are finally formatted and printed on PDF.

Therefore, to establish a mapping between the source code positions in input files and their output elements in PDFs, two essential steps are involved. The first step is to build and attach a position record for every constructed token list when processing input files. The immediate step is to ensure the position record can be propagated to their newly duplicated token lists.

To implement the two steps, it would at first glance appear unavoidable to modify the token list's data structure so that it can store the source code position. However, our patch can avoid this by adopting a significantly different approach. Specifically, the patch hooks two functions including 1) *store_new_token()* and 2) *get_token()*. The former function is used to dynamically allocate the memory space for a new token being appended to a token list. The latter function is used to retrieve a token from memory for duplication. These functions are now augmented. The memory addresses and contents of the newly allocated token or the recently retrieved token will be delivered to the external bookkeeping

program via inter-process communication channels. Meanwhile, if a token is constructed when parsing input files, the corresponding source code position is also delivered.

By monitoring the memory data received, the external program now can keep track of every token list and its associated position record. The maintained information will then be utilized in the hooked *ship_out* function at the PDF output stage. Specifically, if a token list is printed on PDF, the external program will simultaneously reveal the associated position information, which may be stored in the following approaches.

With the enhanced LaTeX engine, every PDF element is now associated with its source code position. The remaining question lies in storing the position information such that a viewer can easily retrieve it. A first solution is to follow SyncTeX, which takes advantage of an auxiliary file. Specifically, the file serves as a dictionary that maps the coordination of PDF elements to their line numbers; whenever a user clicks a specific PDF element on the viewer, the mouse coordinates can be checked against the auxiliary file to obtain the corresponding line number. The principle can also be easily adapted to our work. However, one drawback of this solution stems from the specifically-designed auxiliary file, which can only be parsed by a unique external program [74]. It negatively impacts the portability of PDF files. Alternatively, we propose a novel approach which directly embeds the position meta information to the PDF files. This approach's merit is that it allows viewers to retrieve the positions without the auxiliary file and extern program. Nevertheless, the implementation entails an essential challenge that the PDF specifications do not consider source code synchronization. There exists no generally-acknowledged approach to store the position information on a PDF file.

To tackle this issue, we explore an idea that piggybacks the position information on undocumented PDF metadata. Specifically, PDF metadata is utilized to describe properties such as the font, shape, and position of a PDF element. Among them, a property named *flatness* is barely documented by the PDF specification [4]. We investigate how this property is handled via cross-checking various open-source PDF viewers such as Poppler[46] and Evince [48]. The results indicate that all of the viewers simply ignore this property

when rendering a PDF on screen. In other words, the flatness property can be stored any value without impeding the viewers' functionalities. Therefore, we piggyback the position information on the flatness property. In this manner, whenever a user clicks a PDF element, its flatness property can be retrieved and utilized by a viewer to infer the source code position. Note that since almost every PDF viewer possesses graphical or programming interfaces to access the metadata directly, neither an external program nor a patch for viewers is required. Despite its promising feature, we have to admit that the piggyback approach indeed contains a minor drawback; the resulting PDF's size is slightly bloated due to the extra flatness property. Nonetheless, this side effect is negligible thanks to the PDF stream compression mechanism [4], which greatly reduces the position information's redundancy. Moreover, the piggyback option can be easily switched off when the user decides to publish the final version of the PDF

5.5.3 Faithful Conversion from PDFs to HTMLs

To implement such a WYSIWYG editor, our system has to support not only faithful display, but also dynamic modification of PDF contents (i.e., instantly generating the preview version in response to the user's input). However, it is considered challenging because a PDF document is essentially a vector graphic, which is difficult to edit. Though there exist a limited number of commercial closed-source products (e.g., FoxIt PhantomPDF viewer [44]) which support simple 'touch up' operations such as adding or removing words, we argue that these solutions are not directly applicable for e-paper devices. First, they are originally designed for PCs and tend to pose unacceptable computational burden on resource-constrained e-paper devices. Moreover, these products are generally not portable among devices with different hardware specifications and software architectures.

To address these issues, we explore a novel approach to implement the WYSIWYG view by converting PDFs to HTML pages in a nearly faithful manner. Displaying HTML pages brings about a wide range of advantages. First, HTML pages are of high portability and can be displayed in various devices equipped with browsers. Meanwhile, rendering HTML pages in modern

browsers is a relatively lightweight operation even for devices with low-end specifications. Most importantly, unlike vector-based PDFs, HTML pages are semantic-based; texts in HTML pages are typically surrounded by HTML semantic markup, which allows them to be dynamically modified with the help of Javascript. It facilitates the implementation of commonly-used word processing functionalities (e.g., text selection, insertion, deletion, copying and pasting) in the WYSIWYG view.

To achieve a nearly faithful conversion, SwiftPad employs the following mechanisms to preserve the following kinds of elements in PDFs.

Images. PDF supports graphical instructions such as drawing and image embedding. To preserve the graphic elements, we rasterize them into bitmap images, which then can be displayed using the CSS sprite technique [119].

Font Embedding. Fonts can be embedded in a PDF file, which ensures that readers always see the text in its original font. To preserve the fonts in a PDF, we extract all the fonts from the PDF and convert them into web open font types (WOFF) with the help of two third-party libraries MuPDF [91] and FontForge [138]. The converted fonts then can be embedded in HTML pages using a CSS rule named 'font-face' [119].

Text Locations. Text elements are positioned with absolute coordinates in a PDF document. To preserve the locations in the HTML pages, one naive method is to convert the locations into CSS absolute position rules and assign them to HTML text elements. However, since each text segment is now associated with a unique CSS rule, the page may be too bulky to store and transfer. Moreover, the absolute positions are not flexible; when a user inserts or deletes some texts, a large number of CSS position rules must be changed accordingly in order to achieve text reflow functionality.

Inspired by a more recent tool [132], we attempt a relative positioning method to position each text segment. A simple example in Fig. 5.9 demonstrates the intuition of our approach. Specifically, we first attempt to merge PDF text segments to text lines based on their geometric metrics. Afterwards, we measure the space width between words in each line and turn them into CSS rules. Finally, these rules can be used to construct HTML spacer elements (i.e., empty span elements in a certain width) to help position each text element from left

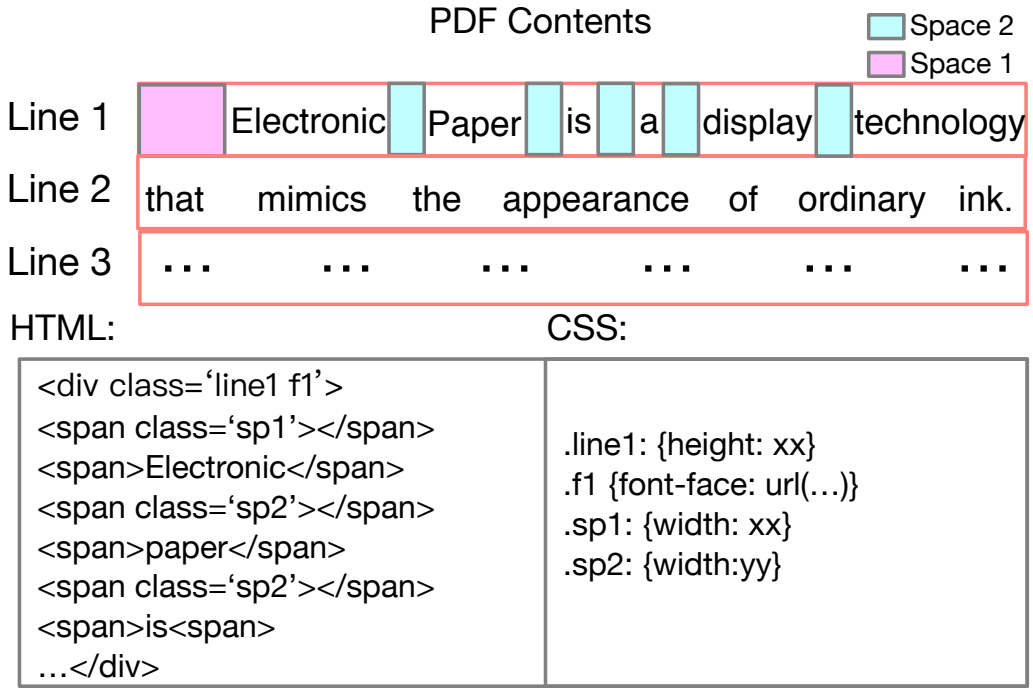


Figure 5.9: PDF to HTML conversion

to right in each line. The advantage of this approach is that the number of generated CSS rules tends to be tiny since there are usually limited number of spacers with different widths in a PDF page. We can further reduce the number by merging some rules which have nearly identical width values at the cost of faithfulness in text locations. As a result, this approach can generate HTML pages with a much smaller size compared with the naive approach. Another important advantage is that this positioning method automatically enables text reflow functionality to a certain extent when texts are inserted or deleted.

5.6 Making SwiftPad Responsiveness

One essential requirement for WYSIWYG editors is high responsiveness. It should allow users to instantly see the final result while a document is being edited. To achieve this goal, SwiftPad employs the following strategy. When

a user types a key, our editor provides immediate response in the sense that it generates and shows a preview version of the document by mimicking the typesetting behavior used in the LaTeX engine. Meanwhile, our editor periodically replaces the current preview version with the latest compilation result from the LaTeX engine. However, such a strategy's effectiveness is negatively impacted by the long compilation time of LaTeX engine. A complication process on conventional engines, depending on the input source codes' complexity, can take several seconds to complete even on a computer with decent specifications (e.g., Intel i7 6900 with DDR4 memory and SSD storage). As a result, users may have to wait a noticeable timespan to view the faithful output.

We find that the inefficiency of conventional LaTeX engines may be attributed to the following two reasons. First, the LaTeX engine, which was programmed decades ago, does not utilize the modern machines' multi-threading feature. Thus, its execution speed is bounded by the clock rate of a single CPU core. Secondly, LaTeX is a batching system, which implies that every time a compilation is initiated, the LaTeX engine must process the input files from the very beginning. Such behavior is undesirable, considering that, in most cases, users only append or modify characters located at the end of the input file, while leaving the preceding contents unchanged. Therefore, recompiling the unchanged contents leads to a considerable amount of repeated and unnecessary computation.

One potential approach to accelerate the complication process is to overhaul the source code of the LaTeX engine and add multi-threading support. However, this requires extensive reworking of the engines. More importantly, it may introduce unheeded bugs undermining software stability. Instead, we shift our attention to the second cause mentioned above and seek a method to avoid the repeated computation between consecutive compilations. The philosophy we apply here is called checkpointing, which consists of saving a snapshot of an application's state such that it can restart from that point in the future. In the context of our enhanced LaTeX engine, we create checkpoints periodically (e.g., after outputting a PDF page) and markdown the corresponding input file positions during the compilation process. When a new compilation job is submitted, based on the file position that users edit on, the enhanced engine

can determine the closest checkpoint and directly start from that point to skip the repeated computation. Another optional trick to optimize responsiveness is that the engine does not have to process the whole input file in most cases. Instead, it may stop right after generating the page the user is currently viewing or editing. By combining this trick with the checkpoint technology, we can ensure the engine's response time remains nearly constant regardless of the page number of documents. For instance, when the user is working on page 5, the engine can start from the checkpoint generated on page 4, and only re-typeset page 5. Though in some rare situations, contents from page 6 onwards may slightly affect the typesetting results of page 5. We argue that the infrequent loss of faithfulness is negligible and would not seriously impact the overall user experience.

We can implement the checkpointing for our engine on three different levels: virtual machine, kernel space, and userspace. On the virtual machine level, we can directly leverage the snapshot functionality of BrowserVM. In other words, we can dump the states of a virtual machine into a disk file. Alternatively, we can also utilize some kernel space checkpointing technologies. It relies on specially-designed kernel modules (e.g., [52]) to save or restore process-related data structures in the kernel. However, the existing in-kernel implementations do not focus on upstream compatibility. As a result, it is difficult to integrate them into recent mainline kernels. To solve the issues of the in-kernel implementations, CRIU [42] proposes a userspace approach; it implements as much checkpointing functionality as possible in the user space and solely uses available kernel interfaces. However, all the above solutions are over-powered and heavyweight for our use case. For example, they checkpoint a wide range of system-wide information not utilized by the LaTeX engine (e.g., TCP sockets and process trees). Each checkpointing operation can take multiple seconds to finish.

These issues motivate us to propose a lightweight userspace checkpoint technology for the LaTeX engine. Specifically, we utilize the hooking technique [41] to patch the engine on the runtime with the checkpoint logic. Each checkpoint operation solely saves three types of information, including 1) CPU registers (e.g., Instruction Pointer), 2) memory segments including Data, Bss, and Heap, 3) current file position for each open file.

A special mechanism is required for memory segments to recover the state of the LaTeX engine fully. At first glance, restoring memory segments seems straightforward; we could simply write back the dump file contents to the original memory addresses. However, it is highly likely to fail due to the stateful nature of the heap segment. Unlike data or bss segments whose sizes are pre-determined during the compilation, the heap segment's size can frequently vary during runtime. Specifically, when an executable allocates/releases a memory block via the standard C library calls 'malloc'/'free', these functions internally invoke a system call 'sbrk' to extend/shrink the heap segment. To efficiently manage the heap segment, the standard C library must keep track of a vital system state, namely, the current heap segment size. When the program is recovering from a checkpoint, this heap segment size must be restored to ensure the C library to continue functioning correctly. However, the restoration functionality is generally not available in most runtime C libraries (e.g., libc).

To address this issue, we patch the built-in heap allocator in the runtime C library such that rather than *sbrk*, it now uses *mmap* as the primary mechanism to obtain memory from the system. The advantage of using mmap is that it allows us to create a heap memory region assigned a direct byte-for-byte correlation with a filesystem object. By duplicating the file, we will obtain a snapshot of the heap memory region. To restore the region, we could map the snapshot file back to memory using *mmap* again. This approach ensures that the heap segment contents and the heap segment size can be correctly restored across runs.

Special care is also given to the file objects, which are kernel objects and do not persist between runs. To address this issue, we will re-initialize every file object by reopening them and restore their file pointer positions by using the function *fseek*.

5.7 Optimizing the E-paper Display Driver

Existing e-paper devices are mainly designed for displaying static content. As a result, they are not well optimized for most office software applications,

which requires constant screen updates. In this section, we demonstrate our enhancement to the e-paper display driver to address this issue.

5.7.1 How E-paper Screens Work?

Before we can demonstrate our enhancement, we first explain the internal display mechanism of e-paper screens. An e-paper screen is controlled by a specifically-designed circuit, the Electrophoretic Display Controller (EPDC). It is responsible for driving corresponding electrical signals to the e-paper panel to update the screen contents upon receiving display update commands from the CPU. Most update commands can be described as a tuple (x, y, w, h, *data*, *mode*). Specifically, the first five parameters denote the coordinate, size, and content of the rectangle region pending to be updated. The *mode* parameter denotes the update mode, whose possible options include 2, 4, 8, or 16 gray levels. The 16 gray level update mode delivers the best display quality (i.e., highest contrast and little ghost effect) but consumes the longest timespan to finish (approximately 1 second). The 2 gray level update mode enables fast animation of the screen contents (approximately 150 ms) but may generate low contrast text and significant ghost effects.

To communicate with the EPDC, the OS requires a kernel driver. A critical task of the driver is to composite an update request tuple. The system can obtain the first five parameters (i.e., x, y, w, h, *data*) directly from the upper layer of the OS. The update mode info is typically not available since its a unique concept for e-paper screens, and most OSs are only designed for LCD screens. Therefore, the drivers have to provide a mechanism to infer the update mode. Most existing driver implementations possess a somewhat rudimentary update mode selection mechanism. One of the most widely-used mechanisms is providing a kernel interface to allow an application to specify the global update mode manually. For instance, an application featuring constantly varied contents can enforce the 2 gray level update mode, while a reading application requiring high display quality can enforce 16 gray level update mode. However, this mechanism is relatively coarse-grained and may not be suitable for applications that simultaneously require fast response time and high display quality, so does our software.

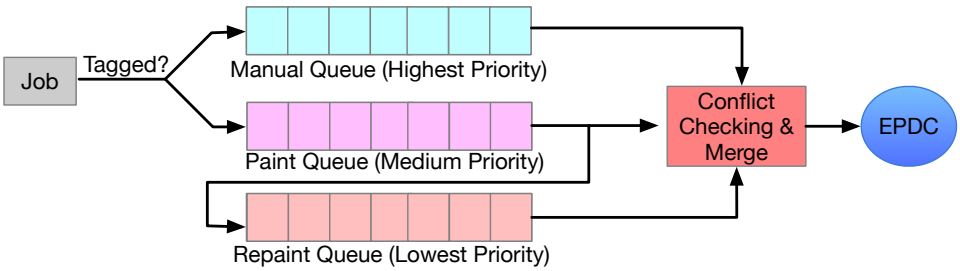


Figure 5.10: Multi-Queue scheduling algorithm

5.7.2 Heuristic Multi-queue Scheduling Algorithm for E-paper Display

In our work, we optimize the existing e-paper display driver to automatically strike a balance between the display quality and response time with a heuristic multi-queue scheduling algorithm.

As shown in Fig 5.10, our algorithm maintains three separate update queues, including a manual queue, a paint queue, and a repaint queue. The manual queue is a simple queue that only stores the incoming jobs with update modes specified by the developer. It is mainly designed to provide backward compatibility for existing e-paper applications.

Most of the time, developers do not manually specify the update mode. In this case, the paint and the repaint queue will be used. Specifically, the paint queue stores all the incoming update requests without update mode info from the application, which will then be sequentially drawn on the screen using 2 gray level mode. Intuitively, this ensures the e-paper screen displays the latest contents as soon as possible (i.e., to deliver a responsive user experience). After leaving the paint queue, the job will enter the repaint queue and be later repainted using 16 gray level mode to improve the display quality (i.e., to alleviate ghost effects and show high-contrast text). In other words, if no new update jobs are submitted to the driver, the high display quality of the screen contents will eventually be achieved.

This algorithm also takes advantage of the simultaneous update feature of the EPDC, which allows multiple update requests to be processed at the same

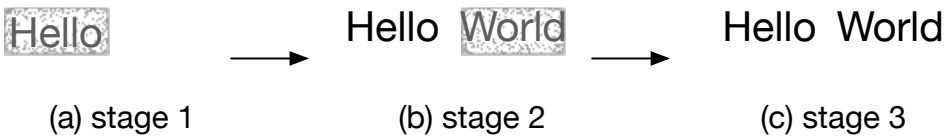


Figure 5.11: How our algorithm takes advantage of the simultaneous update feature of the EPDC.

moment when their update regions do not overlap with each other. In other words, the painting jobs and repainting jobs may be carried out simultaneously if their update regions do not collide. This condition frequently holds for most office software. A typical example is word processors, as depicted in Fig. 5.11. At stage one, the user types the word 'hello', drawn in 2 gray level mode. At stage two, the user types another word 'world', which is still drawn in 2 gray level mode while the previous word 'hello' can be repainted using 16 gray level mode at the same time since the update regions of the two words do not collide. At the last stage, the user stops typing, so the word 'world' is repainted, and all the words are now painted in 16 gray level mode.

Note that the max number of simultaneous update regions is decided by available hardware resources of the EPDC and may be quite limited. When the application is overloading the EPDC by submitting too many update requests at the same time, we prioritize the three queues to optimize the average screen response time (manual >paint >repaint). For an update request in a queue to be executed, two conditions must be met; 1) the EPDC has available resources, 2) all the update requests in those higher priority queues should have either finished the execution or been scheduled (i.e., being drawn). It is also worth pointing out two minor heuristic implementation details designed to pursue faster screen response time. Firstly, our algorithm will attempt to detect and merge the colliding update requests in a queue. Secondly, whenever an incoming request enters the paint queue, existing tasks in the repaint queue colliding with the new request (if any) will be removed.

5.8 Chapter Summary

In this paper, we present SwiftPad, a novel document composition system for e-paper. It integrates the LaTeX typesetting engine with the WYWIWYG editing concept, enabling users to directly edit and instantly review documents in their final print layout. We have identified fundamental architectural challenges and named the corresponding workable solutions. We use SwiftPad to demonstrate how Browserify facilitates cross-platform application development.

Chapter 6

Conclusion and Discussion

This thesis presents Browserify, a framework that enables consistent and rapid application deployment across heterogeneous mobile platforms. Browserify addresses the interoperability and performance issues in existing cross-platform development frameworks via two main components: BrowserVM and Wasmachine. BrowserVM is designed to run unmodified operating systems and applications in browser environments. Wasmachine empowers cross-platform WebAssembly binaries to be efficiently and securely executed in devices with constrained resources. To demonstrate the capabilities of Browserify, we also provide a case study called SwiftPad, where we design a cross-platform document composition system on e-paper devices. Nevertheless, Browserify is an on-going research project and still possesses certain inadequacies. We will suggest potential future research directions in the following sections.

6.1 Future Research on BrowserVM

BrowserVM empowers the execution of unmodified operating systems and applications inside browsers. Nevertheless, BrowserVM is still a research prototype and possesses certain limitations. We are planning to improve BrowserVM in the following aspects.

QEMU IR vs. LLVM IR. BrowserVM is built on top of the QEMU virtualization framework. The framework provides an intermediate representation layer called Tiny Code Generator (TCG) [14]. TCG was initially designed as

a generic C compiler but later made its way to dynamic binary translation. TCG transforms source languages via a TCG frontend to TCG intermediate operations. The operations are then transformed into target instructions via a TCG backend. The frontend performs a limited number of optimizations such as liveness analysis and trivial constant expression evaluation, while the backend usually performs no optimizations. This design improves the speed of dynamic binary translation, but it usually renders the translated blocks suboptimal.

Recently, we are exploring the possibility of replacing the TCG with the LLVM compiler infrastructure [72]. LLVM provides a similar architecture to the TCG. An LLVM's frontend can convert source languages into intermediate LLVM bytecode. The bytecode can then be translated to target machine codes via an LLVM backend. The backend leverages a wide range of optimization techniques (e.g., Peephole optimizations, Loop optimizations, and Link-time optimization) to generate more efficient binary objects at the cost of longer computational times. Nevertheless, it may be worth paying the cost if the translated objects are frequently executed.

Multi-thread Support WebAssembly was first released as a Minimal Viable Product (MVP). MVP only has a small feature-set deemed necessary to make WebAssembly usable. One valuable feature that was missing in the MVP was threading. Recently, IT practitioners and academia are actively experimenting with threads in browsers. Chrome was the first browser to provide experimental threading support.

Recently, we are enhancing multi-core emulation in BrowserVM. BrowserVM currently supports multi-core emulation in a somewhat preliminary manner. Specifically, we use a vCPU to represent a single execution core of the emulated system. Multiple vCPUs are run in turn in a round-robin manner in a single browser worker thread. The browser thread has to constantly perform context switching among vCPUs and likely becomes a performance bottleneck. To address this issue, we are attempting to offload vCPUs to different browser worker threads. We can leverage two newly available APIs called *SharedArrayBuffer* and *Atomics*. They have enabled developers to make use of shared memory across multiple worker threads. Specifically, A SharedArrayBuffer object is used to represent a generic, fixed-length raw binary data buffer. When a SharedAr-

rayBuffer is posted from one worker thread to another, instead of copying the array's content, a reference is sent. As a result, multiple workers have visibility on a shared chunk of memory. A side effect to the memory in one thread will eventually become visible in the other thread. To achieve synchronization, Atomics APIs can be used.

SIMD Support SIMD (Single Instruction Multiple Data) instructions are an essential class of instructions in modern CPUs. They can exploit data parallelism in applications by simultaneously performing the same operation on multiple data elements [59]. Many compute-intensive applications such as video encoding and image processing take advantage of SIMD instructions. Unfortunately, WebAssembly MVP still lacks SIMD instructions, and the WebAssembly SIMD proposal is still at an early stage [43]. A direct consequence is that when translating SIMD instructions, BrowserVM has to un-vectorize the instructions; in other words, BrowserVM transforms them back into non-parallel Single Instructions, Single Data (SISD) instructions. This un-vectorizing process may incur a significant performance penalty. BrowserVM plans to rework the translation engine once SIMD instructions become available in WebAssembly.

Live Migration Live migration refers to the process of moving a running virtual machine between different physical machines without disrupting its running applications. Live migration is useful as it allows a BrowserVM user to offload some running applications from one mobile device to another. It can also be used to implement checkpointing functionality.

BrowserVM supports live migration in a somewhat straightforward manner. When a migration process is initialized, BrowserVM first freezes the virtual machine. BrowserVM then dumps CPU registers, RAM data, and hard disk data into a Javascript Blob file. A different hypervisor can then use the file to reconstruct an identical virtual machine and replicate application states. The running applications may not even notice the interruption. However, one shortcoming of this approach is that the generated snapshot files tend to be bulky. To achieve efficient live migration, state-of-the-art hypervisors such as VMware or VirtualBox apply a more advanced delta snapshot technique. Briefly speaking, the first created snapshot serves as the baseline snapshot, holding the entire content of the virtual disk and memory. All subsequent snapshots

are referred to as delta snapshots, containing only changes made since the last snapshot creation. Delta snapshots are significantly smaller and take less time to transmit via the network. To obtain the actual contents of a snapshot, the hypervisor can start from the baseline snapshot and apply the subsequent delta snapshots. BrowserVM plans to incorporate delta snapshots in the future release.

6.2 Future Research on Wasmachine

Wasmachine enables efficient and secure execution of WebAssembly applications in resource-constrained mobile devices. Nevertheless, Wasmachine is still a research prototype and bears certain limitations. In this section, we suggest several future research directions for Wasmachine.

Static Linking vs. Dynamic Loading. Wasmachine statically links AOT-ed binaries to our Rust-based kernel to generate a deployable system image. Static linking is preferred because it can reduce the system image size thanks to link-time optimization. Specifically, the linker pulls all object files together and combines them into one program. Since the linker can see the program's whole workflow, the link can apply various interprocedural optimization forms to the whole program, achieving higher program performance. One standard link-time optimization is dead code elimination. For example, if none of WebAssembly applications invokes networking functions, the linker then can remove the network-related code from the final system image. Another optimization is function inlining across object files, which replaces a function call site with the called function body. Function inlining eliminates the function calling overhead of frequently-used small functions.

However, we admit that by using static linking, we, to some extent, lose flexibility in system management; when users wish to add an application to a device, they have to recompile all the applications and kernel to generate a new system image for deployment. To make the deployment of applications more convenient, we have implemented kernel support for Executable and Linking Format (i.e., ELF loader). ELF loaders allow our kernel to load an AOT-ed WebAssembly application from local storage or network and set up an execution

context for it in an on-demand manner. This support is still at the experimental stage because it involves changes in many kernel components such as virtual memory, process management, and filesystem.

Monolithic vs. Micro-kernel. Wasmachine entirely throws away hardware protection within the kernel. The kernel code and applications are running in Ring 0, and there exists no conventional kernel/userspace boundary. Since applications and the kernel share the same memory space, Our kernel fits more closely into the category of monolithic architecture.

Wasmachine also gets some inspiration from the micro-kernel architecture. One essential merit of a micro-kernel lies in its modularity; a developer can add, remove, or modify system modules and device drivers on runtime without modifying the kernel code. Similarly, Wasmachine allows developers to extend the system in a modular fashion. Many system components (e.g., audio subsystem) or device drivers (e.g., Ethernet cards) can be implemented as standalone WebAssembly applications and dynamically loaded. However, this support is still experimental because we still lack sufficiently efficient Inter-Process Communication (IPC) APIs in our kernel.

Serverless Computing. Wasmachine is designed to execute long-running WebAssembly applications such as a HTTP server or an in-memory database. However, Wasmachine can be potentially used for another computing diagram, namely, serverless computing. Serverless computing platforms enable developers to host single-purpose applications that automatically scale with demand. In contrast to long-running applications, serverless applications have a significantly shorter lifespan; they are only instantiated when called and shut down immediately when finished. Each instance should also be protected from other instances (i.e., having its memory and filesystem segmented). Most existing serverless platforms achieve these goals by utilizing the Linux control group feature [112]. This feature enables one or more processes to be organized into containers, which have their resources isolated and limited by the kernel. These platforms create a control group (also referred to as container) to host a function and destroy the group right after it is finished. However, starting/stopping containers is a heavyweight process for the Linux kernel. Frequent setup/tear-down of containers leads to a significant performance penalty [95].

Wasmachine may be a viable alternative to existing serverless platforms. Unlike containers, starting/stopping an application in Wasmachine is a relatively lightweight process. Wasmachine also provides a similar process isolation level and performance guarantees. We are currently exploring this idea and seeking solutions to several challenges. First, each serverless WebAssembly application needs to be compiled by our AOT compiler before execution. This process may be time-consuming and lead to a long initialization time. Secondly, Wasmachine, as a multi-processing platform, should provide a mechanism to limit an application's execution time and maximum memory usage. Lastly, Wasmachine should also provide a robust web API to manage and scale their applications quickly.

Fine-grained Resource Provision. Wasmachine, as a multi-processing OS, should provide a mechanism to limit an application's execution time and maximum memory usage. To achieve these goals, we plan to learn from Linux and introduce a new system call named *setrlimit*. This system call enables developers to apply a soft resource limit and a hard resource limit on processes. When the soft limit is met, a SIGCPU signal is sent, and the process can decide how to handle it. For example, the process may decide to limit its resource consumption upon receiving the signal. When the hard limit is met, a SIGKILL signal is sent and kills the process without impediments. By using the same system call, we can also enforce a memory limit. The system call can issue an ENOMEM error from a violating sbrk() syscall and forcibly terminate the process.

6.3 Future Directions on SwiftPad

SwiftPad demonstrates the potential of Browserify in building cross-platform applications on resource-constrained devices. Nevertheless, as a document composition system for e-paper, it still bears several limitations that need further improvement.

Faithfulness of the WYSIWYG Editor. Our WYSIWYG editor attempts to maintain visual consistency when converting the compiled PDF into

an HTML file. Nevertheless, we still have to admit that the HTML file rendering results may imperceptibly differ from the PDF in certain aspects.

One imperfection lies in text locations. Specifically, our conversion algorithm utilizes spacers defined by relative positioning CSS rules to position each text element from left to right in each line. To reduce the number of CSS rules, we can optionally merge some rules with nearly identical width values. This approximation can significantly reduce memory usage and improve rendering speed at the loss of the display's quality (i.e., the inaccurate spacing between each word). However, such a tiny difference in spacing is typically not perceptible to human eyes unless users compare the PDF and HTML side by side. The converted HTML file can be considered an output from an old-school dot matrix printer, which may not deliver perfect typesetting details but provides good readability to users.

Besides the text locations, some PDF elements cannot be currently converted into HTML tags losslessly. For example, our current implementation does not support PDF hyperlink, which prevents users from following a link in a table of contents to navigate different PDF pages. Apart from the hyperlink, fillable PDF forms are also ignored and rendered as rectangles. Another example is that our implementation rasterizes all figures embedded in a PDF file, making vector graphics not more scalable. We will attempt to address these issues in our next prototype.

Improving the Performance of E-Paper Screens. Though e-paper devices possess many appealing features such as non-glowing screen and silent fanless design, their most notable drawback, low refresh rate, is still preventing themselves from being widely adopted.

In this work, SwiftPad attempts to combat the drawback in the software level by integrating a heuristic multiple-queue scheduling algorithm to the graphics stacks. In future work, we plan to reformulate the scheduling process as an optimization problem with the queueing theory's help. We may be able to propose an algorithm that computes the optimal scheduling policies. Less ideally, we could design an approximate algorithm with a performance guarantee.

Another potential method to improve the screen response time is to enhance the e-paper display controller, EPDC. Most controllers nowadays rely on a

firmware to function correctly. By modifying the firmware, we may control the e-paper screen in a more low-level manner (e.g., electrical signal or hardware registers) to pursue higher performance. Nevertheless, this approach is challenging to implement because most e-paper manufacturers regard the firmware's design as a commercial secret and reject releasing its source codes. This understandable action inevitably prevents e-paper technology from receiving contributions from various developers. To circumvent this issue, we aim to design our e-paper display controller using Field Programmable Gate Array (FPGA). However, the process could still be challenging. We need to obtain low-level detailed e-paper hardware information, which is seldom documented.

Bring True WYSIWYG Editing in LaTeX. SwiftPad attempts to enable as many as commonly-used WYSIWYG features. One example is to find a natural interface for structural edits. Our preliminary design is a menu displaying predefined LaTeX snippets or formulas, which are examples of the needed elements. For instance, a section header snippet could be "`\subsection{Sectiontitle}`". In the menu, this is shown in its compiled version. If the user selects the snippet, a paste (insert) of the snippet at the current cursor position is performed. This workflow reduces the structural change to a simpler concept, the paste concept. It also provides a natural visual appearance and is end-user-programmable.

Meanwhile, we have developed a WYSIWYG mathematical editor for users to overhaul an entire formula. Right-clicking any math formulas can invoke this editor, and the corresponding LaTeX mathematical expression will be automatically extracted and displayed. Any modification on the expression will be detected, and a preview will be immediately rendered using the MathJax processor [24]. In the editor, a cheat sheet of the LaTeX mathematical symbols is also presented. If a symbol is clicked, it will be inserted into the current mathematical expression. In this manner, the user can edit the expression at ease without remembering all the LaTeX mathematical syntax. Likewise, we also offer users with a graphical LaTeX table generator. The tool allows users to specify the row/column number of tables and alter the properties (e.g., contents, or styles) of each table cell in a spreadsheet-like interface. Once the editing is finished, the generated table can be inserted to a user-specified location.

Nevertheless, we have to admit that bringing full WYSIWYG to LaTeX is

still not entirely fulfilled and may not be an easy task. One essential issue is that commonly-used LaTeX engines such as pdfTeX and XeTeX are mainly implemented in an ancient language, CWEB. Maintaining and extending the legacy source codes typically requires extensive work. What makes matters worse is that we have to maintain the backward comparability of a considerable number of LaTeX packages. This requirement makes a radical change to LaTeX engines less likely to be accepted by the LaTeX communities. In the future, we aim to re-factor the LaTeX engines using modern languages so that developers can make their contributions in a much easier manner.

Exploring Alternative Input Technologies. Currently, SwiftPad adopts keyboards as the main input technology due to its popularity and users' familiarity. Nevertheless, we also realize the potential of other alternative input options, for instance, voice input and handwriting input. SwiftPad currently provides experimental support for these two input options for users with Google's voice API and text recognition API respectively. We plan to conduct a usability study to evaluate each input method's efficiency under different scenarios.

Bibliography

[1] 2018. URL: http://time.com/3682621/this-country-just-made-it-illegal-to-give-kids-too-much-screen-time/.

[2] Nasir Abbas et al. "Mobile edge computing: A survey". In: *IEEE Internet of Things Journal* 5.1 (2017), pp. 450–465.

[3] Aclements. *A 21st century LaTeX wrapper*. 2015. URL: https://github.com/aclements/latexrun/.

[4] Adobe. *Portable Document Format*. 2008. URL: https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000%5C_2008.pdf.

[5] *Adobe PhoneGap: Build amazing mobile apps powered by open web tech.* URL: https://phonegap.com/.

[6] Suyesh Amatya and Arianit Kurti. "Cross-platform mobile development: challenges and opportunities". In: *International Conference on ICT Innovations*. Springer. 2013, pp. 219–229.

[7] Kijin An, Na Meng, and Eli Tilevich. "Automatic inference of java-to-swift translation rules for porting mobile applications". In: *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 2018, pp. 180–190.

[8] *Android Distribution dashboard*. URL: https://developer.android.com/about/dashboards.

[9] *Applause: a toolkit for easily creating cross-platform mobile applications.* URL: https://github.com/applause/applause.

[10] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. "Sourcerer: An internet-scale software repository". In: *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. IEEE. 2009, pp. 1–4.

[11] T Ball. *J2ObjC-Java to Objective-C Translator and Run-Time*. 2015.

[12] Michael Bebenita et al. "SPUR: a trace-based JIT compiler for CIL". In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 2010, pp. 708–725.

[13] Fabrice Bellard. "Jslinux-javascript pc emulator running linux". In: *URL: https://bellard. org/jslinux/tech. html* (2013).

[14] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[15] Brian N Bershad et al. "Extensibility safety and performance in the SPIN operating system". In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), pp. 267–283.

[16] Andreas Biørn-Hansen and Gheorghita Ghinea. "Bridging the gap: Investigating device-feature exposure in cross-platform development". In: *Proceedings of the 51st Hawaii International Conference on System Sciences*. 2018.

[17] Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. "A survey and taxonomy of core concepts and research challenges in cross-platform mobile development". In: *ACM Computing Surveys (CSUR)* 51.5 (2018), pp. 1–34.

[18] Andreas Biørn-Hansen et al. "An empirical investigation of performance overhead in cross-platform mobile development frameworks". In: (2020).

[19] Andreas Biørn-Hansen et al. "An empirical study of cross-platform mobile development in industry". In: *Wireless Communications and Mobile Computing* 2019 (2019).

[20] Karlheinz Blankenbach et al. "22-2: Smart Pharmaceutical Packaging with E-Paper Display for improved Patient Compliance". In: *SID Symposium Digest of Technical Papers*. Vol. 49. 1. Wiley Online Library. 2018, pp. 271–274.

[21] Alan W Brown. *Integrated project support environments: the aspect project*. Vol. 33. Elsevier, 2013.

[22] Antuan Byalik. "Automated cross-platform code synthesis from web-based programming resources". PhD thesis. Virginia Tech, 2015.

[23] Luca Cardelli et al. "Modula-3 language definition". In: *ACM SigPlan Notices* 27.8 (1992), pp. 15–42.

[24] Davide Cervone. "MathJax: a platform for mathematics on the Web". In: *Notices of the AMS* 59.2 (2012), pp. 312–316.

[25] Sanchit Chadha et al. "Facilitating the development of cross-platform software via automated code synthesis from web-based programming resources". In: *Computer Languages, Systems & Structures* 48 (2017), pp. 3–19.

[26] Po-Sheng Chiu et al. "Interactive Electronic Book for Authentic Learning". In: *Authentic Learning Through Advances in Technologies*. Springer, 2018, pp. 45–60.

[27] Matteo Ciman and Ombretta Gaggi. "An empirical analysis of energy consumption of cross-platform frameworks for mobile development". In: *Pervasive and Mobile Computing* 39 (2017), pp. 214–230.

[28] J Collins. "Latexmk 4.43 a: Fully automated LaTeX document generation". In: *Penn State Department of Physics, Pennsylvania State University* (2015).

[29] Ezra Cooper et al. "Links: Web programming without tiers". In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2006, pp. 266–296.

[30] Leonardo Corbalan et al. "Development frameworks for mobile devices: a comparative study about energy consumption". In: *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 2018, pp. 191–201.

[31] Luis Corral, Alberto Sillitti, and Giancarlo Succi. "Mobile multiplatform development: An experiment for performance analysis". In: *Procedia Computer Science* 10 (2012), pp. 736–743.

[32] Jessica M Dauw. "Screen Time and the Effects on Development for Children Ages Birth to Five Years". In: (2016).

[33] Thomas Degueule. "Composition and interoperability for external domain-specific language engineering". PhD thesis. Rennes 1, 2016.

[34] Nick Desaulniers. "Emscripten and WebGL". In: *WebGL Insights* (2015), p. 71.

[35] Ryan Dewsbury. *Google web toolkit applications*. Pearson Education, 2007.

[36] Sunny Dhillon and Qusay H Mahmoud. "An evaluation framework for cross-platform mobile application development tools". In: *Software: Practice and Experience* 45.10 (2015), pp. 1331–1357.

[37] Paul E Dickson. "Cabana: a cross-platform mobile development system". In: *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 2012, pp. 529–534.

[38] Andrew Dillon, Cliff McKnight, and John Richardson. "Reading from paper versus reading from screen". In: *The computer journal* 31.5 (1988), pp. 457–464.

[39] Alan Donovan et al. "Pnacl: Portable native client executables". In: *Google White Paper* (2010).

[40] Aline Ebone, Yongshan Tan, and Xiaoping Jia. "A performance evaluation of cross-platform mobile application development approaches". In: *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE. 2018, pp. 92–93.

[41] Manuel Egele et al. "Dynamic spyware analysis". In: (2007).

[42] P EMELYANOV. *CRIU: Checkpoint/Restore In Userspace*. 2019. URL: https://criu.org/.

[43] *Fast, parallel applications with WebAssembly SIMD*. URL: https://v8.dev/features/simd.

[44] Foxit Reader. *PDF Viewer from Foxit Software*. 2019. URL: https://goo.gl/w1918D.

[45] Bob Frankston. "Progressive web apps [bits versus electrons]". In: *IEEE Consumer Electronics Magazine* 7.2 (2018), pp. 106–117.

[46] FreeDesktop. *Poppler Pdf Rendering library*. 2018. URL: http://poppler.freedesktop.org/.

[47] Abhi Gambhir and Gaurav Raj. "Analysis of cache in service worker and performance scoring of progressive web application". In: *2018 International Conference on Advances in Computing and Communication Engineering (ICACCE)*. IEEE. 2018, pp. 294–299.

[48] GNOME. *Document Viewer*. 2018. URL: https://github.com/GNOME/evince.

[49] Tor-Morten Grønli et al. "Mobile application platform heterogeneity: Android vs Windows Phone vs iOS vs Firefox OS". In: *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*. IEEE. 2014, pp. 635–641.

[50] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In: *ACM SIGPLAN Notices*. Vol. 52. 6. ACM. 2017, pp. 185–200.

[51] Adam Hall and Umakishore Ramachandran. "An execution model for serverless functions at the edge". In: *Proceedings of the International Conference on Internet of Things Design and Implementation*. ACM. 2019, pp. 225–236.

[52] Paul H Hargrove and Jason C Duell. "Berkeley lab checkpoint/restart (blcr) for linux clusters". In: *Journal of Physics: Conference Series*. Vol. 46. 1. IOP Publishing. 2006, p. 494.

[53] Henning Heitkötter, Tim A Majchrzak, and Herbert Kuchen. "Cross-platform model-driven development of mobile applications with md2". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 2013, pp. 526–533.

[54] John L Henning. "SPEC CPU2006 benchmark descriptions". In: *ACM SIGARCH Computer Architecture News* 34.4 (2006), pp. 1–17.

[55] David Herman, Luke Wagner, and Alon Zakai. "asm. js.(2014)". In: *Retrieved October* 7 (2014), p. 2017.

[56] Jason D Hiser et al. "Evaluating indirect branch handling mechanisms in software dynamic translation systems". In: *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE. 2007, pp. 61–73.

[57] David Holman et al. "Flexkit: a rapid prototyping platform for flexible displays". In: *Proceedings of the adjunct publication of the 26th annual ACM symposium on User interface software and technology*. ACM. 2013, pp. 17–18.

[58] Arvind Hudli, Shrinidhi Hudli, and Raghu Hudli. "An evaluation framework for selection of mobile app development platform". In: *Proceedings of the 3rd International Workshop on Mobile Development Lifecycle*. 2015, pp. 13–16.

[59] Christopher J Hughes. "Single-instruction multiple-data execution". In: *Synthesis Lectures on Computer Architecture* 10.1 (2015), pp. 1–121.

[60] Galen C Hunt and James R Larus. "Singularity: rethinking the software stack". In: *ACM SIGOPS Operating Systems Review* 41.2 (2007), pp. 37–49.

[61] *IL2JS: An Intermediate Language to JavaScript Compiler*. URL: `https://github.com/Reactive-Extensions/IL2JS/`.

[62] *Ionic: One codebase. Any platform..* URL: `https://ionicframework.com/`.

[63] Martin Jacobsson and Jonas Wåhslén. "Virtual machine execution for wearables based on WebAssembly". In: *13th EAI International Conference on Body Area Networks (BodyNets 2018)*. 2018.

[64] Abhinav Jangda et al. "Not so fast: Analyzing the Performance of WebAssembly vs. native code". In: *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 2019, pp. 107–120.

[65] Xiaoping Jia and Christopher Jones. "AXIOM: A Model-driven Approach to Cross-platform Application Development." In: *ICSOFT*. 2012, pp. 24–33.

[66] *Jor1k: Online OR1K Emulator running Linux*. URL: https://github.com/s-macke/jor1k/.

[67] Nicholas Kardaras. *Glow Kids: How Screen Addiction Is Hijacking Our Kids-and How to Break the Trance*. St. Martin's Press, 2016.

[68] Wafaa S El-Kassas et al. "Taxonomy of cross-platform mobile applications development approaches". In: *Ain Shams Engineering Journal* 8.2 (2017), pp. 163–190.

[69] Richard Koch et al. "TEXshop". In: *See website at http://www. uoregon. edu/koch/texshop/texshop. html* (2006).

[70] Radhesh Krishnan Konoth et al. "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 1714–1730.

[71] Chris Lattner. "LLVM and Clang: Next generation compiler technology". In: *The BSD conference*. Vol. 5. 2008.

[72] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[73] Jerôme Laurens. "Direct and reverse synchronization with SyncTEX". In: *TUGBoat* 29 (2008), pp. 365–371.

[74] Jerôme Laurens. *Library for Parsing SyncTeX files*. 2011. URL: https://packages.debian.org/sid/libsynctex-dev.

[75] Olivier Le Goaer and Sacha Waltham. "Yet another DSL for cross-platforms mobile development". In: *Proceedings of the First Workshop on the globalization of domain specific languages*. 2013, pp. 28–33.

[76] Doug Lea and Wolfram Gloger. *A memory allocator*. 1996.

[77] In Lee and Kyoochun Lee. "The Internet of Things (IoT): Applications, investments, and challenges for enterprises". In: *Business Horizons* 58.4 (2015), pp. 431–440.

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

[79] John Levine. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.", 2009.

[80] Amit Levy et al. "The case for writing a kernel in Rust". In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. ACM. 2017, p. 1.

[81] *Little Kernel*. URL: `https://github.com/littlekernel/lk`.

[82] Florian Loitsch and Manuel Serrano. "Compiling scheme to javascript". In: ICFP. 2006.

[83] Tim Majchrzak and Tor-Morten Grønli. "Comprehensive analysis of innovative cross-platform app development frameworks". In: *Proceedings of the 50th Hawaii International Conference on System Sciences*. 2017.

[84] Gurmeet Singh Manku and Rajeev Motwani. "Approximate frequency counts over data streams". In: *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier. 2002, pp. 346–357.

[85] Eric Masiello and Jacob Friedmann. *Mastering React Native*. Packt Publishing Ltd, 2017.

[86] Nicholas D Matsakis and Felix S Klock II. "The rust language". In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM. 2014, pp. 103–104.

[87] Iván Tactuk Mercado, Nuthan Munaiah, and Andrew Meneely. "The impact of cross-platform development approaches for mobile applications from the user's perspective". In: *Proceedings of the International Workshop on App Market Analytics*. 2016, pp. 43–49.

[88] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux Journal* 2014.239 (2014), p. 2.

[89] Sebastiano Miano et al. "Creating complex network service with eBPF: Experience and lessons learned". In: *High Performance Switching and Routing (HPSR). IEEE* (2018).

[90] AB MoSync. *MoSync–Cross-platform SDK and HTML5 tools for mobile app development*. 2015.

[91] mupdf. *MuPDF, a lightweight PDF, XPS, and E-book viewer*. 2018. URL: https://mupdf.com.

[92] Marius Musch et al. "New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2019, pp. 23–42.

[93] Nellie Bowles. *Is the Answer to Phone Addiction or a Worse Phone?* URL: https://www.nytimes.com/2018/01/12/technology/grayscale-phone.html.

[94] Cisco Visual Networking. "Cisco global cloud index: Forecast and methodology, 2015-2020. white paper". In: *Cisco Public, San Jose* (2016).

[95] Edward Oakes et al. "{SOCK}: Rapid Task Provisioning with Serverless-Optimized Containers". In: *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 2018, pp. 57–70.

[96] Henry Oswald, James Allen, and Brian Gough. *ShareLaTeX, the Online LaTeX Editor*. 2018.

[97] Overleaf. *Overleaf, Collaborative Writing and Publishing*. 2018. URL: https://www.overleaf.com.

[98] T Petricek and Don Syme. "AFAX: Rich client/server web applications in F#". In: (2007).

[99] Charles Petzold. *Creating Mobile Apps with Xamarin. Forms Preview Edition 2*. Microsoft Press, 2015.

[100] Rob Pike. "The go programming language". In: *Talk given at Google's Tech Talks* 14 (2009).

[101] Boydlee Pollentine. *Appcelerator Titanium Smartphone App Development Cookbook*. Packt Publishing Ltd, 2011.

[102] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. "Seahawk: Stack overflow in the ide". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 1295–1298.

[103] Stefan Poslad. *Ubiquitous computing: smart devices, environments and interactions*. John Wiley & Sons, 2011.

[104] Gilda Pour. "Understanding software component technologies: Javabeans and activex". In: *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No. PR00275)*. IEEE. 1999, pp. 398–398.

[105] Bobby Powers, John Vilk, and Emery D Berger. "Browsix: Bridging the gap between Unix and the browser". In: *ACM SIGPLAN Notices* 52.4 (2017), pp. 253–266.

[106] Arno Puder. "An xml-based cross-language framework". In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2005, pp. 20–21.

[107] Peixin Que, Xiao Guo, and Maokun Zhu. "A comprehensive comparison between hybrid and native app paradigms". In: *2016 8th International Conference on Computational Intelligence and Communication Networks (CICN)*. IEEE. 2016, pp. 611–614.

[108] Lyle Ramshaw. "Eliminating go to's while preserving program structure". In: *Journal of the ACM (JACM)* 35.4 (1988), pp. 893–920.

[109] Micha Reiser and Luc Bläser. "Accelerate JavaScript applications by cross-compiling to WebAssembly". In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. ACM. 2017, pp. 10–17.

[110] Research and Markets. *e-Paper Display Market 2016-2020*. 2018. URL: https://goo.gl/PvVHTs.

[111] Christoph Rieger and Tim A Majchrzak. "Weighted evaluation framework for cross-platform app development approaches". In: *EuroSymposium on Systems Analysis and Design*. Springer. 2016, pp. 18–39.

[112] Rami Rosen. "Linux containers and the future cloud". In: *Linux J* 240.4 (2014), pp. 86–95.

[113] Marija Selakovic and Michael Pradel. "Performance issues and optimizations in javascript: an empirical study". In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 61–72.

[114] Suranga Seneviratne et al. "A survey of wearable devices and challenges". In: *IEEE Communications Surveys & Tutorials* 19.4 (2017), pp. 2573–2620.

[115] Rahul Sharma, Vesa Kaihlavirta, and Claus Matzinger. *The Complete Rust Programming Reference Guide*. 2019.

[116] Eva Siegenthaler, Pascal Wurtz, and Rudolf Groner. "Improving the usability of E-book readers". In: *Journal of Usability Studies* 6.1 (2010), pp. 25–38.

[117] Eva Siegenthaler et al. "Comparing reading processes on e-ink displays and print". In: *Displays* 32.5 (2011), pp. 268–273.

[118] Andreas Sommer and Stephan Krusche. "Evaluation of cross-platform frameworks for mobile applications". In: *Software Engineering 2013-Workshopband* (2013).

[119] Steve Souders. "High-performance web sites". In: *Communications of the ACM* 51.12 (2008), pp. 36–41.

[120] Adam Tacy et al. *GWT in Action*. Manning Publications Co., 2013.

[121] Sajjad Taheri et al. "OpenCV. js: Computer Vision processing for the open Web platform". In: *Proceedings of the 9th ACM Multimedia Systems Conference*. ACM. 2018, pp. 478–483.

[122] Watanabe Takuya and Hidehiko Masuhara. "A spontaneous code recommendation tool based on associative search". In: *Proceedings of the 3rd International Workshop on search-driven development: Users, infrastructure, tools, and evaluation*. 2011, pp. 17–20.

[123] Aneesh P Tarun et al. "PaperTab: an electronic paper computer with multiple large flexible electrophoretic displays". In: *CHI'13 Extended Abstracts on Human Factors in Computing Systems*. ACM. 2013, pp. 3131–3134.

[124] *The edge cloud platform behind the best of the web*. URL: `https://www.fastly.com/`.

[125] Stefan Tilkov and Steve Vinoski. "Node. js: Using JavaScript to build high-performance network programs". In: *IEEE Internet Computing* 14.6 (2010), pp. 80–83.

[126] Abhishek Tiwari et al. "A large scale analysis of android—Web hybridization". In: *Journal of Systems and Software* (2020), p. 110775.

[127] Gianluca Tosini, Ian Ferguson, and Kazuo Tsubota. "Effects of blue light on the circadian system and eye physiology". In: *Molecular vision* 22 (2016), p. 61.

[128] *V86: x86 virtualization in JavaScript, running in your browser and NodeJS*. URL: `https://github.com/copy/v86`.

[129] Kenton Taylor Varda et al. *Cloud computing platform that executes third-party code in a distributed cloud computing network*. US Patent 10,331,462. June 2019.

[130] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. "Stackoverflow and github: Associations between software development and crowdsourced knowledge". In: *2013 International Conference on Social Computing*. IEEE. 2013, pp. 188–195.

[131] John Vilk and Emery D Berger. "Doppio: breaking the browser language barrier". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014, pp. 508–518.

[132] Lu WANG and Wanmin LIU. *Online publish via pdf2htmlEX*. 2013.

[133] John M Wargo. *Apache Cordova 4 Programming*. Pearson Education, 2015.

[134] *Wasmer: a WebAssembly Runtime*. URL: `https://wasmer.io/`.

[135] *WebAssembly Specification*. URL: `https://webassembly.github.io/spec/core/index.html`.

[136] Elliott Wen, Jim Warren, and Gerald Weber. "PaperWork: Exploring the Potential of Electronic Paper on Office Work". In: *Proceedings of the ACM Symposium on Document Engineering 2019*. 2019, pp. 1–10.

[137] Elliott Wen and Gerald Weber. "SwiftPad: Exploring WYSIWYG TEX Editing on Electronic Paper". In: *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2020, pp. 1–4.

[138] George Williams. "Font creation with FontForge". In: *EuroTEX 2003 Proceedings, TUGboat* 24.3 (2003), pp. 531–544.

[139] Michiel Willocx, Jan Vossaert, and Vincent Naessens. "A quantitative assessment of performance in mobile app development tools". In: *2015 IEEE International Conference on Mobile Services*. IEEE. 2015, pp. 454–461.

[140] A Wingo. "JavaScriptCore, the WebKit JS implementation". In: *Oct* 28 (2011), pp. 1–12.

[141] Spyros Xanthopoulos and Stelios Xinogalos. "A comparative analysis of cross-platform development approaches for mobile applications". In: *Proceedings of the 6th Balkan Conference in Informatics*. 2013, pp. 213–220.

[142] Bennet Yee et al. "Native client: A sandbox for portable, untrusted x86 native code". In: *2009 30th IEEE Symposium on Security and Privacy*. IEEE. 2009, pp. 79–93.

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

[144] Alon Zakai. “Fast Physics on the Web Using C++, JavaScript, and Emscripten”. In: *Computing in Science & Engineering* 20.1 (2018), pp. 11–19.

[145] B van der Zander, J Sundermeyer, D Braun, et al. *TeXstudio*. {`https://www.texstudio.org/`}. 2018.