Toward a Unified Framework for the Heterogeneous Cloud Utilizing Bytecode Containers

by

David Andrew Lloyd Tenty

Bachelor of Science (Honours), Ryerson University, 2016

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the program of

Computer Science

Toronto, Ontario, Canada, 2019

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my dissertation may be made electronically available to the public.

Toward a Unified Framework for the Heterogeneous Cloud Utilizing Bytecode Containers

Master of Science 2019

David Andrew Lloyd Tenty

Computer Science

Ryerson University

Abstract

As we approach the limits of Moore's law the Cloud computing landscape is becoming ever more heterogeneous in order to extract more performance from available resources. Meanwhile, the container-based cloud is of growing importance as a lightweight way to deploy applications. A unified heterogeneous systems framework for use with container-based applications in the heterogeneous cloud is required. We present a bytecode-based framework and it's implementation called Man O' War, which allows for the creation of novel, portable LLVM bitcode-based containers for use in the heterogeneous cloud. Containers in Man O' War enabled systems can be efficiently specialized for the available hardware within the Cloud and expand the frontiers for optimization in heterogeneous cloud environments. We demonstrate that a framework utilizing portable bytecode-based containers eases optimizations such as heterogeneous scaling which have the potential to improve resource utilization and significantly lower costs for users of the public cloud.

Acknowledgements

I would like to express my heartfelt thanks to my supervisor Dr. Ali Miri for all his support, patient guidance and advice. Thank you as well to my committee for the detailed feedback that has made this a better work. A sincere thanks to the system administrators of the Department of Computer Science for all their help over the years and for entertaining my wild ideas.

Thank you to all the peers who have helped and cheered me along the way, my fellow students in our lab, the course union and many others around the campus. A big thanks to my comrades at CUPE Local 3904 for their encouragement and picking up the slack while I was away writing.

Thank you to all my dear friends who helped me along this path, especially Michael Biggs and Adrian Popescu for many interesting and fruitful discussions and their help proof reading the drafts.

Thank you to my family for your endless love and support. To my soon-to-be parents and sister in-law for your kindness, prayers and patiently waiting for this work to finish so the wedding could begin. To my fiancée, Amira Rezkalla, for being my love, my rock and saving me from myself throughout this process. To my father, Boudewijn, for setting me on your lap in front of a computer all those years ago, beginning a journey that continues to this day. And to my late mother Adriana for choosing the video game console over the bike, I wish you could see how far I've come.

Dedication

Dedicated to my fiercest supporters:

my fiancée Amira

&

my mother Adriana

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As we approach the limits of Moore's law the landscape of computing is becoming ever more heterogeneous in order to extract more performance from available resources [42]. This includes the cloud computing landscape with the widespread adoption of parallel accelerators such as GPUs and new processor types such as Amazon's ARM-based AWS Graviton processor [23].

On the software side, the container-based cloud is of growing importance in modern cloud computing due to containerization's popularity as a light weight way to virtualize applications [27]. For example, Cloud-Native Applications (CNA), applications which are formed from a "distributed, elastic and horizontal scalable system composed of (micro)services ... and operated on a self-service elastic platform" [15] are increasingly being built using popular container tools such as Docker. Correspondingly this self-service elastic platform is increasingly container-based and Google Cloud Platform's Kubernetes Engine [9], Amazon's Elastic Container Service [1], and Microsoft's Azure Kubernetes Service [2] are all public cloud services based on containers that have launched in the last few years.

As customers of these computing platforms are typically charged by their usage of the underlying computing resources, as per the Utility computing model, efficient usage of those resources is an important object of study in cloud computing and container research [27]. Cloud providers are also naturally interested in providing a mixture of resources that will appeal to their customers while minimizing operational costs. This work will examine how the architectural and mircoarchitectural diversity of computing resources can be exploited in heterogeneous clouds to improve performance and resource utilization.

However, there are barriers both logistical and technical to container-based applications being able to easily migrate and adapt to optimally take advantage of this heterogeneous environment. Heterogeneous systems present a great deal of complexity in the form of varying capabilities and requirements to potential developers who wish to exploit their benefits, necessitating a great deal of software development effort. Heterogeneous system frameworks seek to alleviate this by presenting a single coherent interface to heterogeneous computing devices.

In order to deal with the full-range of heterogeneity available in the container-based cloud, a new

Figure 1.1: A Portuguese Man O' War

framework is required. We will introduce the concept of bytecode-based[1] container images in order to present the basis for a novel framework and our implementation we call Man O' War, named after the Portuguese Man O' War (shown in Figure 1.1): a "jelly-like marine invertebrate, where clones bud and can have specialized function within the colony" [43]. This system allows for the creation of portable bytecode-based containers for use in the heterogeneous cloud. Much like in their namesake, containers in a Man O' War enabled container system can be easily adapted and specialized for the particular location they are needed within the cloud and they are adapted specific node hardware they will be running on. After describing the system, some example use cases, and its integration into a container environment, we discuss performance experiments conducted with the system and a discussion of dynamic optimization strategies and larger cloud-wide opportunities enabled by the system such as Heterogeneous Scaling. We demonstrate these optimizations can result in dramatic increases in cost efficiency for some workloads. We conclude with potential future directions for research and extensions to the system.

In this chapter we will discuss containers, the container-based cloud and the role of containers in the modern software development ecosystem. We will move on to discuss the nature of the heterogeneous computing resources that are currently available in the cloud and the optimization opportunities

[1]Bytecode is used colloquially to a variety of low-level virtual Instruction Set Architectures (ISAs).

they present. We will then conclude with a discussion of the difficulties in meshing container-based development approaches with the optimization opportunities found in the heterogeneous cloud.

1.1 Overview

Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [20].

1.1.1 Cloud Definition

The NIST Definition of Cloud Computing (SP 800-145) is a useful reference model for our discussions. In it Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [20].

This cloud model is composed of five essential characteristics, three service models, and four deployment models [20]. We briefly reproduce the essential characteristics here as we will refer to them in our later discussion:

Essential Characteristics

On-demand self-service A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

Broad network access Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g. mobile phones, tablets, laptops, and workstations).

Resource pooling The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.

Rapid elasticity Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

Measured service Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service

1.1.2 Containers in the Modern Cloud

Virtualization of resources is essential in the cloud to delivering on pooling and sharing resources. Containers are a type of operating system-level virtualization which are implemented by a number of popular technologies such as Docker[2], provide an isolation environment including a complete base filesystem image for a process or collection of processes to run in. Containers can be a high-performance, low-overhead alternative to virtual machines in the modern cloud[35].

The container image is composed of several filesystem snapshots called layers. It is managed by a container engine which, upon a request to create an instance of the container, unpacks it and launches an initial process within the environment with access to only the outside resources as described by the manifest in the image. An image contains an application with all of its dependencies packed into one convenient distributable artifact.The same image unaltered image can be built, tested in the development and then deployed directly to production on container-based infrastructure and platforms. This setup has numerous advantages and within popular movements such as DevOps containers as the actual software artifacts produced and consumed have become very popular given their ability to isolate the application and it's dependencies from its underlying environment [7].

1.1.3 The promise of heterogeneity in the cloud

A heterogeneous system consists of processing elements of varied type and capability, presenting unique optimization opportunities from a systems perspective. These processing elements may be part of the same logical machine accessible through a local bus or part of a distributed system with nodes physically and possibly geographically remote, such as in the public cloud. The key characteristic in these systems of study is that we wish to leverage the system's heterogeneity to improve some performance characteristic of our applications, such as total execution time, response time, power efficiency, or cost. While such systems have in practice existed for many years, for example in the High Performance Computing (HPC) domain, recently interesting results have become available that show the opportunities for optimization in public cloud infrastructures [31].

[2]Docker is a very popular tool for construction and running container environments developed by the company of the same name.

1.2 Scope and Goal

1.2.1 Difficulties in exploiting heterogeneity

Making use of the heterogeneous computing resources in the cloud in practice is complicated by a number of factors. Even in the absence of a unifying framework in order to be able to leverage cloud-level heterogeneity our program must on some level be portable. By portability we mean simply that the program is capable, generally without extraordinary measures such as complex emulation, of being run in a different environment. We also must distinguish two types of portability: source and binary. In the case of source portability, which is indeed the goal of many modern programming languages and environments such as POSIX [11], the source code of the program is portable between platforms. Binary portability, which implies source portability, means that the final compiled representation of the program is portable. The range of environments a program is portable for affects the types of optimization we can perform for the heterogeneous environment.

Containers are portable between container runtimes and host environments. Containers are not however portable between operating systems and hardware architectures as they generally contain platform specific binaries which are not portable. According to Open Container Initiative (OCI) standards, container images and filesystem layers are normally tagged with the particular hardware architecture they are built for. Even if the original application is source portable, this means rebuilding and optimizing the container for each possible node type.

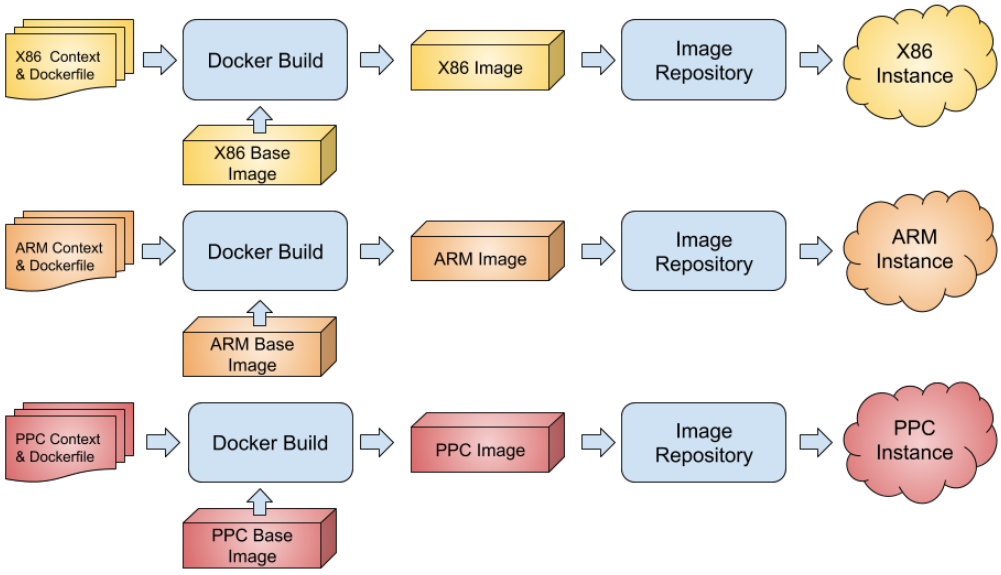


Figure 1.2: A Docker multi-architecture build and deployment

Figure 1.2 shows what a high level overview of a typical build and deployment process might look like using Docker if a developer wished to target instances with three different architectures: x86_64 (x86),

ARM, and POWER (PPC). Note that the builds are entirely separate even if we are building the same software, with the same options, as the base image and/or executables packaged into the context will need to be architecture specific. In the best case, even if we are packaging an application written in a language which itself is portable (eg. Python) this still means packaging a platform specific version of its runtime environment.

More problematically, even on the same architecture, for any possible variation of microarchitecture the container developer wishes to optimize for, they must build specialized binaries and a corresponding container image for those resources. Consider an example where we wish to utilize processor extensions including Single Instruction Multiple Data (SIMD) instruction sets which can exploit parallelism, for example Advanced Vector Extensions (AVX). In this case we will need to ensure our containerized application is rebuilt to support it. Since many container images are built and published by ISVs to registries such as Docker Hub, to be composed with other application layers they may omit these types of optimizations entirely, so we are required to rebuild our application and all of its dependencies. On the opposite end of the spectrum, container-based PaaS such as Amazon Fargate offer no choice of heterogeneity at all meaning we are stuck with a one cost/performance point fits all situation.

While these types of platform and portability woes are certainly not new, we feel they are breaking some of the abstractions and portability gains that containers introduce. One of the primary advantages of container-based cloud is decoupling the application from the infrastructure. However once we attempt to perform heterogeneous performance optimizations this quickly starts to break down. A overlying heterogeneous systems framework will allow us to regain some sense of a unified model.

Clearly we will require a new, portable basis for containers as well in our unified framework if we wish to be able to easily make use of the available heterogeneity in the cloud. As part of this work we will introduce a binary architecture portable, bytecode container format and our implementation of this toolchain called Man O' War, which allows containers to be translated into a form appropriate for whichever hardware architecture they may be run on. This allows us to move towards a unified heterogeneous systems framework while still producing containers that may be run unmodified on existing container infrastructure as required.

1.3 Our Contributions

While we have shown that in practice it is possible to benefit from exploiting heterogeneity in cloud systems [31], such systems are often used in a limited capacity in cloud systems; some times limited solely to the use of accelerators such as Graphics Processing Units (GPUs) [31] and rarely do we encounter a system-wide focus on enabling flexibility as well as allowing for optimization in the presence heterogeneity. A unified approach and heterogeneous systems framework for use with container-based application virtualization in the presence of a potentially heterogeneous cloud environment is required.

In the rest of this work we introduce our solution for the construction of such a framework and demonstrate our implementation on scenarios presenting optimization difficulties using container-based systems in the heterogeneous cloud. We show it is possible to enable simplified heterogeneous node

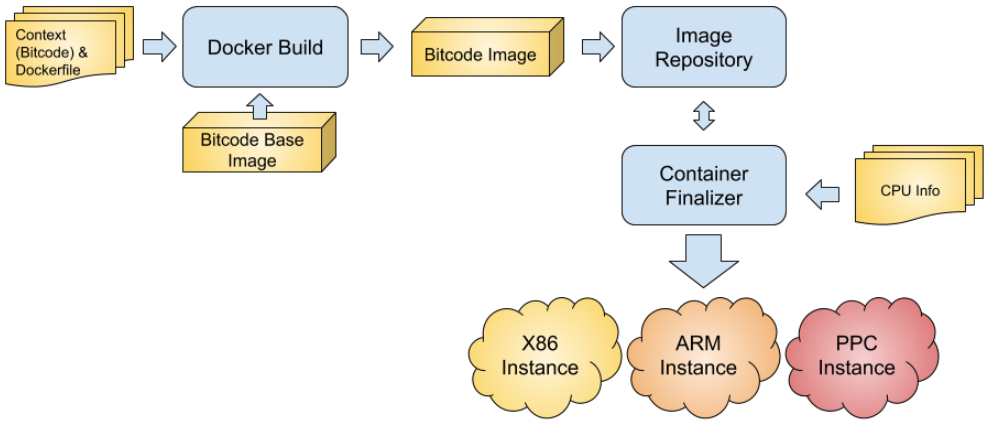


Figure 1.3: Man O' War bytecode container build and deployment

support by building bytecode-based program translation into the foundation of these platforms. We introduce Man O' War, a system based on the LLVM[3] bitcode[4] format and apply it to the Open Container Initiative (OCI) container format to produce a portable containers suitable for use with unmodified OCI compliant runtimes. A high level overview of the build and deployment process using Man O' War containers is shown in Figure 1.3 (compare with Figure 1.2). We propose methods for incorporating these new containers into existing container or orchestration system by utilizing a translation system called a finalizer which will translate and optimize images to the available node hardware.

With this groundwork laid, we discuss large-scale optimizations that are possible in the heterogeneous cloud that can take advantage of the flexibility afforded by a heterogeneous environment such as Heterogeneous Scaling. We demonstrate via an experiment that in case of the real world heterogeneous clouds by incorporating nodes of diverse architecture/performance it is possible to achieve significant cost savings on some workloads. We perform experiments that show that they offer access to specific performance advantages owing to their in heterogeneous hardware environments due to their dynamic ability to exploit hardware specific optimization and as well there is generally no notable performance overhead to using portable bytecode containers over traditionally compiled, single architecture containers so they maybe utilized universally.

1.4 Organization of this work

The rest of this work is organized as follows:

Chapter 2 will discuss background and introduce some related works we draw upon and that are necessary for an understanding of our approach.

[3]Originally LLVM stood for Low-Level Virtual Machine[18], though in current usage it is no longer an acronym.
[4]Bitcode is usually used for the LLVM bytecode, we use the terms interchangeably when referring to our own LLVM-based implementation.

In chapter 3 we will describe how bytecode techniques can help us solve the challenge of a unified heterogeneous systems framework for cloud and the Man O' War bytecode container system we developed. We will continue with a discussion of how this system addresses the potential barriers to performance optimization in the heterogeneous cloud.

In chapter 4 we will conduct a number of experiments with the Man O' War system which demonstrates its practicality in real cloud system and scenarios. We also do a detailed comparison and discussion with some related works.

Finally we wrap up in chapter 5 with our conclusions, a discussion of potential future work such as various dynamic optimization methods which are made possible by bytecode containers and which can be incorporated into container-based clouds, and finally other future directions and extensions we have considered.

Chapter 2

Background and Related Works

In this chapter we discuss the background of heterogeneous systems, as well as bytecode techniques and the container-based cloud that are important for our problem in the subsequent chapters. We will also discuss several related works and projects that are important for understanding of our work and solution. We will return to many of the systems describe here later in the discussion section of our evaluation chapter.

2.1 Heterogeneous Systems

2.1.1 Introduction and Definition

A heterogeneous system is a coherent system consisting of multiple types of processing elements combined to achieve greater scalability, energy efficiency, and performance than is achievable with a single type of processor [32]. Some examples may include:

- General Purpose Processors (eg. CPU)
- Parallel Accelerators (eg. GPU)
- Reconfigurable Systems (eg. FPGA)

All of these processing elements may vary greatly in architecture and microarchitecture. They may have radically differing memory models with different consistency models. The system may function in either a local or distributed fashion. In addition to physical processing elements we may choose to incorporate virtualized ones as well. This great variability is the strength of heterogeneous systems, but also introduces some new challenges that we will consider in the following sections.

2.1.2 Frameworks for Heterogeneous Systems

Managing the complexity of really world heterogeneous systems necessitates a framework and/or runtime to address the software engineering challenge of running programs on these highly variable systems.

Without such a framework the software developer would need to address all the variability of the programming models of the various devices as well as orchestrating the resulting computation across the devices.

From the paper and presentation on the heterogeneous systems framework "Dandelion" by Rossbach et al.[32], which we will discuss in more detail later in this chapter, we obtain their description of goals for an idealized framework for heterogeneous systems. Primarily this is that it should provide a single programming interface for:

- CPU
- GPU
- FPGA
- Other current and future devices ("You name it")

It should allow the programmer to as much as possible write simple sequential code while the framework or runtime adopts all responsibility for:

- Parallelizing the computation
- Partitioning data
- Running on all available resources
- Mapping the computation to the best architecture

These are lofty and unrealizable goals for developers of real systems to aim for[1] and they are forced to compromise on a subset of these or other limitations such as targeting a specific set of programs or devices [32]. Two important sub-problems can be identified from the above description of the problem which are identifying the mapping (scheduling) and running on all available resources (portability).

Scheduling

The problem of mapping the computation to available devices is fundamentally an optimization and scheduling problem. Let us consider our own simplistic way of modelling the problem of running a job given such a system, so we can understand better what is involved.

Given a set of processes $P_1 \dots P_n$ and a set of computing resources $R_1 \dots R_m$, we must find a plan or schedule S where S maps all processes P to resources R subject to a cost function C which measures the overall cost (eg. time, throughput, power, monetary cost)[2] of running all the P processes with the given R resources and schedule S. Thus, we are trying to find solutions to the following optimization problem:

[1] Dandelion also does not attempt to meet their idealized definition by their own admission[32].
[2] Roloff *et al.* discuss a family of such cost functions based on node cost and execution-time [31].

$$Argmin \sum C(P, R, S) \tag{2.1}$$

That is we are trying to find the minimum cost way to run the processes with the given choice of resources. In practice this problem is also subject to a number of constraints; some are design requirements such as scheduling deadlines and performance requirements[3], others are limitations such as program restrictions or resource limits. These limits are of key importance as they restrict our ability to place processes in their optimal locations.

This is an instance of the classical job shop scheduling problem. While this optimization problem is in general intractable, it admits numerous interesting approximations and scheduling in heterogeneous systems whether local, distributed, virtual, cloud or otherwise is a well studied problem [41][14][28][31][33]. All the heterogeneous systems frameworks we consider later address this problem in some way, whether explicit to the user or not.

Portability and Program Representation

An ideal framework must attempt to ensure the maximal possible computation resources that are available for use can be utilized without regard to the specific underlying type or architecture of those resources. While complete abstraction of all types of resources is perhaps unrealistic, this gives rise to the concept of portable representations of the program. A program written with the framework should be able to run on all the different physical and virtual devices supported by that particular framework. The means of achieving this vary by framework and context, but this will be a recurring and central theme for the rest of the discussion of frameworks in this chapter as well as the rest of our work.

2.1.3 Example Frameworks

In this section we discuss some important instance of frameworks for heterogeneous systems. This list is by no means exhaustive[4], but it is illustrative of the types of frameworks that exist and several that are important and related to our own work.

OpenCL

OpenCL is an open standard and framework for implement access to parallel accelerators maintained by the Khronos Group and originally authored by Apple [38]. It allows developers to access the capabilities of parallel accelerators by writing programs called 'kernels' in a C like language with support for expressing parallel operations, which is compiled from source by the device driver and uploaded to the device upon execution. The framework contains functionality for managing the kernels execution and marshaling data to and from the accelerator [38].

Newer versions of OpenCL have been implemented on top of a bytecode called SPIR-V, solving the portability problem with an intermediate language that it shares with other Khronos projects and which

[3]Performance requirement such as to maintain a Service Level Agreement (SLA).

[4]We do not discuss CUDA the popular NVIDIA framework for example, as it is proprietary and not used for our implementation.

allows pre-compiled versions of the kernels to be shipped with the application and later translated on demand [37].

Heterogeneous System Architecture (HSA)

The Heterogeneous System Architecture (HSA) is a heterogeneous framework and associated compliant hardware that allows different types of processors to work together efficiently and cooperatively through shared memory developed by the HSA Foundation with broad industry participation [42, p.2]. Their motivation includes that Central Processing Unit (CPU) have "reached a plateau in computational speed per watt" and thus many hardware platforms are becoming heterogeneous, incorporating non-traditional processing devices such as GPUs and Digital Signal Processor (DSP) which can exploit the parallelism inherent in many applications [42].

They note that the traditional architectural approach of handling these devices as I/O devices means that significant overheads arise in software that handles the task initiation and data movement, meaning developers must make sure tasks are substantive enough to warrant offloading. A second major issue of this approach is that the programming interface of these devices require a programmer to provide explicit compute kernel, often written in a completely separate language, that they must then explicitly interface with their main program. This adds significant overhead to the software development process and can disrupt the overall architecture of applications.

HSA aims to enable seamless and efficient co-operation of these different types of processors through a shared memory construct. It provides a unified programming interface to heterogeneous systems that consists of diverse parallel processors from multiple vendors. They define heterogeneous system architecture intermediate language (HSAIL) "a low-level compiler intermediate language, designed to express parallel regions of code and be portable across multiple vendor platforms and hardware generations"[42, p.19].

Compilers supporting the platform are intended to take traditional high-level languages with parallel extensions and compile them into HSAIL [42, p.20]. The resulting HSAIL is designed to be embedded in a binary format (BRIG) inside a standard executable along side native code in a type of Fat Binary. Later it is retrieved and passed to the HSA runtime system to be compiled by a finalizer (either Ahead-of-Time (AOT) or Just-in-Time (JIT) on the host). Once the program begins executing it can be dispatched to available device queues in an HSA-enabled operating system. Front-end languages for compilations include OpenCL, C++ AMP (a set of parallel computing extensions) and others [42].

Dandelion

In "Dandelion: a Compiler and Runtime for Heterogeneous Systems" Rossbach *et al.* present a project which tries to address the programmability challenge for data-parallel programs on heterogeneous systems [32]. It uses the .NET LINQ (Language Integrated Query) framework to embedded data-parallel operations encoded in a relational type syntax in a general purpose programming language. The compiler and runtime then take care of automatically distributing the data and tasks over CPU, GPU and FPGA backends [32].

They identify two primary challenges for the framework. Firstly, the heterogeneity of the system must be well encapsulated from the programmer and present them with a simple and familiar programming model in the presence of this diversity. Secondly, the system must attempt to efficiently integrate the underlying runtimes to achieve high performance [32].

In order to run the same code in different architectural contexts, they introduce a new general-purpose cross compiler framework for .NET bytecode that allows it to be translated to multiple back-end devices [32]. This allows the programmer to write code in a mostly familiar way. They believe this prototype shows the viability of using rich object-oriented languages for programming data-parallel computing on heterogeneous systems [32].

2.2 Bytecode and Intermediate Representations

Several of the heterogeneous systems frameworks we have describe above make use of various types of bytecodes and it merits discussing bytecodes in more detail. Bytecodes are a low-level machine independent representation for programs. The origins of bytecode systems date back to the era of o-code and BCPL [4]. They have long been used as a machine independent intermediate representation for programs in a compiler system. Bytecodes can either be compiled directly to machine code (e.g. LLVM) or executed by a system such as a process virtual machine (e.g. Java Virtual Machine (JVM)) which consists of a software implementation of the virtual architecture utilized by the bytecode.

Many, many different bytecodes and related systems using them exist for different purposes and systems. We have already mention a few in our discussion of frameworks for heterogeneous systems. We will discuss just a few in this section that are especially relevant to our work and problem.

2.2.1 Relevant Bytecode Systems

LLVM

LLVM is a framework, a bitcode format and open-source project originally discussed in the 2004 paper "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation" by Latter and Adve [18]. It consists of a standardized bytecode representation that serves as a common representation for analysis, transformation, and code distribution as well as a compiler framework that makes use of these features to perform advanced optimizations [18]. We will give a brief overview and discuss a few details of these, for further reading we recommend the wealth of excellent documentation available on the LLVM project website.

The LLVM representation describes a program in a RISC-like instruction set that has three interchangeable formats: a textual assembly-like syntax, a compact on-disk binary format (known as bitcode), and in-memory format (for use by the compiler system). The instruction set contains many advanced features which aid analysis, for example it is presented in Single Static Assignment (SSA) format where registers are taken from an infinite set, must be defined before use and are assigned exactly once. It has a type system consisting of fixed-sized primitive types, four derived types (i.e. pointers, arrays, struc-

tures, and functions) as well as an explicit cast operator. It has an explicit load/store architecture for memory access as well as the instruction *alloca* for stack allocation. It also has an explicit function call instruction which abstracts away platform calling conventions and explicit instructions for implementing exceptional control flow (e.g. exceptions)[18].

Let us consider a brief example of a typed bitcode instruction. The *getelementptr* instruction is an important instruction used in accessing arrays and structures that we will refer back to later in our discussions. It used by the LLVM system:

> "to perform pointer arithmetic in a way that both preserves type information and has machine-independent semantics. Given a typed pointer to an object of some aggregate type, this instruction calculates the address of a sub-element of the object in a type-preserving manner (effectively a combined '.' and '[ ]' operator for LLVM)."

In terms of the compiler system itself, static compiler front-ends for the source languages will produce the LLVM representation, which is then linked together and Link Time Optimization (LTO) is performed. After the any optimization passes are perform the code is then translated into native-code for a particular target and the LLVM code is store with the native code[5].

In the paper they set out the following advantages of the LLVM system:

1. persistent program information
2. offline code generation
3. user-based profiling and optimization
4. transparent runtime model
5. uniform, whole-program compilation

The first of these, *persistent program information*, is provided by the unified LLVM representation that can be persisted through out the programs lifetime to perform advanced optimizations at all stages, compile-time, link-time, install-time, run-time, etc. Secondly, *offline code generation* means we can run advanced optimizations that are too costly to do at runtime[6]. *User-based profiling and optimization* means that the LLVM framework can insert instrumentation to help profile application behaviour and use those profiles as the basis to tune program optimizations[7]. *Transparent runtime model* means the system isn't tied to any particular object model, runtime environment or other aspect of a particular high-level language, allowing any language to be compiled using it and finally, *Uniform, whole-program compilation* means that this language agnostic nature allows us to optimize whole programs as one unit after linking (what is often called LTO).

[5]This is the description in the original paper, however we note that many actual LLVM-based compiler toolchains do not store the bytecode

[6]This is in contrast to what is done when using JIT techniques in some virtual machines

[7]We will return to the subject of profiling-based optimization in our final chapter.

In terms of limitations, they note that they do not intend for LLVM to be a universal compiler IR, since it cannot represent transformations that depend on high-level language features. They consider it complementary to virtual machine techniques because LLVM has no notion of high-level constructs, does not depend on a particular runtime system, and "does not guarantee type safety, memory safety, or language interoperability any more than the assembly language for a physical processor does" [18]. LLVM is a very low-level representation and even C has features that must be lowered by the compiler targeting LLVM. LLVM bitcode is also not inherently portable as a frontend may embed a great deal of platform/target specific information into our bitcode depending on the construction of the language and tools being utilized.

2.2.2 Portable Native Client

Native Client (NaCl) is a now discontinued[8] open source project from Google [17] with the aim of allowing native C and C++ programs to be run on and integrated into the web platform. While we are interested specifically in Portable Native Client (PNaCl) bytecode that was later added to the system, we will give a brief description of the original system to aid overall understanding of the context of that work.

In "Native Client: A Sandbox for Portable, Untrusted x86 Native Code" Yee *et al.* note that the browser environment tends to make certain sets of computations infeasible for browser-based apps due to performance constraints and it's limitation to JavaScript-based software limits the use of many libraries and other code but the alternative of remote execution of native code presents a wealth of security concerns [45]. They introduce Native Client "a constrained execution environment for native code to prevent unintended side effects and a runtime for hosting these native code extensions through which allowable side effects may occur safely"[45]. This infrastructure allows hosting x86 binary modules in an OS and browser-portable manner[9].

The application is prohibited from accessing the native operating system's system call interface and is constrained to an inner sandbox via x86 segmented memory. A trusted service runtime and interface in the processes address space provides access to system services (roughly analogous to a system call interface) and a simple datagram-based message service allows it to communicate to JavaScript and other components in the browser (similar to a microkernel). The browser implements an API known (in later versions) as Pepper, which allows them to ship a C standard library implementation that relies on the browser for most functionality that would be found on a typical OS. This provides a high level of separation from the operating system. The rest of their implementation consists of a modified GCC-based toolchain and a validator that ensures the resulting binary has certain security properties before it is loaded. They performed validation on the SPEC2000 benchmark suite and show that their approach is limited to on average a 5% performance penalty over all benchmarks compared to directly running native code.

[8]Native client has been depreciated with the adoption of Web Assembly (WASM) as a cross-browser standard for compiled applications running on JavaScript Virtual Machines (VMs).

[9]Only Google Chrome actually supported Native Client, a probable factor in it's downfall.

After the initial release Google released an extension to Native Client called Portable Native Client (pNaCl) which introduced a bytecode approach which attempted to alleviate some of it's short comings [6]. They introduce the Portable Native Client executable:

> "While the operating-system neutrality of Native Client tends to encourage good practices with respect to ISA portability, the burden of building, testing and deploying a program on all supported hardware platforms—currently IA-32, ARM and x86-64—lies with the developer. This arrangement makes it too easy for the developer to fail to support one or more ISAs, and tends to create a barrier for future new ISAs, threatening the portability promise of the Web." [6]

They note that this requires a great deal of developer effort that may be beyond the reach of some smaller developers and temps them to make the invalid simplifying assumption "all the world's an x86" [6]. This is the exact situation we have with heterogeneous systems, and PNaCl is key to our own approach to portability problems.

PNaCl uses LLVM bitcode as an ISA-neutral representation for it's binary format and provides a translator (based on LLVM) that generates the native code on the client side as part of the web browser where the CPU architecture is known. The resulting binary is then fed to the Native Client runtime as per normal.

2.2.3 JIT Technology with C/C++

In their 2013 paper "JIT Technology with C/C++: Feedback-Directed Dynamic Recompilation for Statically Compiled Languages" Nuzman *et al.*[24] discuss how advanced users, especially those in HPC, can heavily optimize their code "so the most up-to-date capabilities of the hardware can be unleashed" but more generally this is a rarity:

> "More commonly, developers, users building open-source software, and Independent Software Vendors (ISVs) building software in production do not enable the highest optimization levels, nor do they use hardware specific optimizations or Feedback-Directed Optimizations (FDOs). The implications of such highly optimizing build processes on software production costs are such that only a single-step build with moderate optimizations is actually used in practice." [24]

Their further motivating example from a vendor's, such as IBM's, perspective is the difficulty hardware and tools vendors encounter with the large gap between when the introduce new hardware features and when ISVs actually begin enabling those features in their software, thus making those benefits available to end users of these applications running on their hardware. They go on to describe how dynamic optimization, using information such as the exact CPU model that becomes available later, can help close this performance gap by enabling efficient use of processor features and removing this burden from the software development process.

Their bytecode-based solution is a runtime recompilation system for statically compiled languages based on Fat Binaries with optimization driven by dynamic profiling [24]. It is based on generating Fat Binaries which contain both a platform native binary version of the program and an Intermediate Representation (IR) version from an IBM in-house split compilation toolchain based on the XL C compiler (they comment that this approach can also be perform with LLVM or similar systems)[24]. The dynamic optimization is provided at runtime by a JIT infrastructure taken from the IBM Testarossa Java compiler infrastructure. They validate their results on the SPECint2006 benchmark suite, demonstrating that the same profiling and instrumentation can be leveraged largely as is from a Java JIT compiler and that including a binary version of the program can offset the startup costs of JIT for a solely IR version of the program such as experienced with some virtual machines.

2.3 Cloud Computing and the Container-based Cloud

We introduced the definition of cloud based on the NIST model in chapter 1 and in this section we expand on and discuss specifics of the container-based cloud. As we have noted containerization is a popular approach to building cloud systems based on operating system level virtualization that makes up the domain for our investigation and implementation.

As mentioned in our introduction, virtualization of resources is essential to delivering on the *resource pooling* nature of the cloud. Over time during the growth and development of the cloud differing types of virtualization have become popular. System level virtualization in the form of hypervisors and system virtual machines has been a dominant type of virtualization for computing resources in the cloud. In recent years however operating system level virtualization known as containerization has become a popular alternative due to the increase resource efficiency of such techniques [35]. A number of application virtualization studies have shown that containers are a high-performance, low-overhead alternative to virtual machines [35] and accordingly we see containers used increasingly in both Infrastructure as a Service (IaaS) scenarios in the cloud as alternative to hypervisors and as a Platform as a Service (PaaS) with cloud-hosted orchestration systems [27].

Container-based application virtualization radically simplifies application deployment, configuration and dependency management. It is consistent with and essential for achieving the *On-demand Self Service* and *Rapid Elasticity* characteristics of the cloud as defined by NIST [20], and thus containerization has become one of the common ways to deploy applications in the modern cloud and Container-based systems occupy both the IaaS and PaaS portion of the cloud stack [27].

Linux Containers, an implementation of the containerization concept for the Linux OS is implemented by a number of popular technologies such as LXC [19] and Docker [5] among others. It provides an isolation environment including a complete base filesystem image for a process or collection of processes to run in. Separation of the processes running within a container from the outside environment is normally accomplished through a kernel isolation mechanism called cgroups. The container image, which may be composed of several filesystem snapshots called layers, is managed by a container engine which, upon a request to create an instance of the container, unpacks it and launches an initial process

within the environment with access to only the outside resources as described by its manifest.

Container orchestrators such as Kubernetes[10] tie a number of nodes running these container engines into a container-based cloud, by managing container lifecycle events such as creation and destruction and provisioning of resources such as networking and storage volumes on a cluster-wide scale.

This setup has numerous advantages from a software engineering perspective. A container image contains an application with all of its dependencies packed into one convenient artifact. Within popular movements such as DevOps, which stress the need for automation such as Continuous Integration (CI) and Continuous Delivery (CD), containers as the actual software artifacts produced and consumed in these processes have become very popular given their ability to isolate the application from its underlying environment [7]. An unaltered, identical image can be tested in the development pipeline, then deployed directly to production on container-based infrastructure, as well as scaled to as many instances as required in keeping with desire for *Rapid Elasticity*. Recently the Open Container Initiative (OCI)[26] has been established to standardize container formats and runtime environments in the name of interoperability, and Docker has contributed large parts of their container runtime to that effort [26].

Containers have also become very popular for performance reasons, as a number of application virtualization studies have shown that containers are a high-performance, low-overhead alternative to virtual machines [35]. Accordingly we see containers used increasingly in both IaaS scenarios in the cloud as alternative to hypervisors and as a PaaS with cloud-hosted orchestration systems [27].

We have discussed heterogeneity and heterogeneous systems, bytecode-based systems and the container based cloud. In the next chapter we will introduce the difficulties and opportunities encountered when heterogeneity and the container-based cloud intersect and our specific problem as well as our solution.

[10]Kubernetes is a container orchestrator originally developed by Google and released as open source.

Chapter 3

Unified Framework for the Heterogeneous Cloud Utilizing Bytecode Containers

In this chapter we first take a look in detail at the barriers we wish to solve on the way to incorporating heterogeneity into the container-based cloud. We then continue on to discuss the potential of a heterogeneous systems framework using bytecode-based solutions to address some of these problems and introduce the Man O' War system, our contribution for generating portable bytecode containers for use within heterogeneous cloud environments. We discuss it's various components including virtual ABI, toolchain, standard library, finalizer and container format and conclude by discussing how this and similar systems may be integrated into existing container-based environments.

3.1 Preliminaries

3.1.1 Types of Heterogeneity in the Cloud

"Central Processing Units (CPUs) have reached a plateau ... As a result, all computing systems, from mobile devices to supercomputers, are quickly becoming heterogeneous" [42, p.1]. Heterogeneity in the modern cloud comes from several sources.

Firstly, on the individual node scale. For example, Nemirovsky, Markovic, Unsal and Cristal consider the case of heterogeneous and homogeneous processors:

> "Whereas initial chip multiprocessors (CMPs) integrated several identical computation cores per chip, known as homogeneous processors, we now see an increasing tendency to explore the integration of diverse computational cores, called heterogeneous processors" [22].

Often this means in cloud instances at the hardware level we see systems that include heterogeneous

cores such as designs with Non-Uniform Memory Access (NUMA) or parallel accelerators such as GPU. Also at the software level, cloud providers may offer instances that are tuned to different performance characteristics such as high memory bandwidth or having a low latency configuration [31].

Secondly, while these heterogeneous processing elements are of interest to us, we may aim to consider the problem of exploiting heterogeneity between different nodes in the cloud beyond what is available on a single host. As put by Roloff *et al.*, "building a cloud system out of more than one instance type is an area that has been researched less. A system composed of different instance types is considered a heterogeneous cloud"[31]. What this means in practice is when provider(s) make these diverse types of computation resources available to us, perhaps by offering different combinations of hardware/software features to tenants at different prices (for example offering low-power and high performance processor instances), we are in a heterogeneous cloud environment and new optimization opportunities arise.

3.1.2 Advantages of the Heterogeneous Cloud

In their paper "Heterogeneity-aware adaptive auto-scaling heuristic for improved QoS and resource usage in cloud environments" Sahni and Vidyarthi describe how most auto-scaling mechanisms in cloud are restricted to a single type of server configuration, even though the plethora of configurations and prices available mean the ability to scale by different sizes can provide greater elasticity and cost/resource efficiency [33].

They identify what they call "heterogeneous scaling" in cloud which allows for scaling across different sized VMs [33]. We will utilize an even more generalized definition where heterogeneous scaling is scaling across all different types and sizes of computational devices available in the modern cloud. If we are able to incorporate the full range of heterogeneous systems, even higher elasticity and resource efficiency can be achieved.

In "Exploiting Price and Performance Tradeoffs in Heterogeneous Clouds" Roloff *et al.* explore the use of multiple instance types in the heterogeneous cloud to improve cost efficiency, reducing the price of execution while maintaining a similar application performance [31]. Their "results show that heterogeneous clouds are able to execute parallel applications with a reduced cost, while maintaining a similar performance as homogeneous clouds"[31].

They begin by defining a metric called the cost-delay product CDP shown in Equation 3.1 (roughly proportional to the cost efficiency) that provides a basis for analyzing cost-vs-performance tradeoffs in the heterogeneous cloud. They define two other metrics in this family, C^2DP and CD^2P, which take the squares of these respective quantities in order to emphasize the impact of either lower cost or higher performance.

$$CDP = cost\ of\ execution\ \times\ execution\ time \tag{3.1}$$

They perform an evaluation on the Microsoft Azure cloud using the NAS Parallel Benchmarks using OpenMPI. They compared the benchmarks running across all combinations of 8 nodes selected from D4 and F8 node types. They computed the CDP-family metrics for each of the configurations and reported the cost efficiency gains across all configurations.

Their results show that a heterogeneous mixture of resources is most efficient for the majority of benchmarks. Also, each possible configuration is most efficient for some particular benchmark:

> "This shows that simple homogeneous clouds that do not take the specific application behavior into account can not result in optimal cost efficiencies."[31]

These results show some of the very important performance optimization opportunities presented by the heterogeneous cloud. In some cases they were able to improve the cost efficiency of the system by 42.3%[31]. We use their metrics later in our own performance experiments (laid out in chapter 4) regarding nodes with heterogeneous architecture, as the nodes used in this study are all the same hardware architecture, and demonstrate more exciting opportunities.

3.1.3 Difficulties in exploiting heterogeneity

Using containers as a platform for making use of the heterogeneous computing resources in the cloud in practice is complicated by a number of factors, many of which we have discussed in our introduction in section 1.2.1 which we discuss again here. One of primary concern is that while we might be able to utilize a parallel accelerator through an existing heterogeneous systems framework but in order to truly be able to leverage cloud-level heterogeneity our program must on some level be portable. Portability means in this case that the program is capable, generally without extraordinary measures such as complex emulation, of being run in a different environment.

While Containers are portable between standardized container runtimes, with the absences of a unifying framework however containers are not however portable between operating systems and hardware architectures as they generally contain platform specific binaries which are not portable. Refer to the multi-layered build process in figure 1.2 that shows what a high level overview of a typical build and deployment process might look like using Docker if a developer wished to target instances with three different architectures. Even if we are packaging an application written in a language which itself has some portability, such as an interpreted language, this still means packaging a platform specific version of its dependencies and runtime environment. More problematically, even on the same architecture, for any possible variation of microarchitecture the container developer wishes to utilize and optimize for, they must build specialized binaries and a corresponding container image for those resources.

As we stated before, we feel that these barriers are breaking some of the abstractions and portability gains that containers introduce. Decoupling the application from the infrastructure is among the primary advantages of container-based cloud and the cloud in general. However once we attempt to perform performance optimizations this quickly starts to break down. On the opposite end of the spectrum, container-based PaaS such as Amazon Fargate offer no choice of heterogeneity at all meaning we are stuck with a one cost/performance point fits all situation. We will require a new, portable basis for containers if we wish to be able to make use of the available heterogeneity in the cloud without extensive developer effort.

3.2 Toward a Unified Framework for the Heterogeneous Cloud

As we have seen it is well established that heterogeneous systems, whether using mixed CPU types and system configurations, accelerators such as GPUs or even reconfigurable computing elements such as FPGAs are beneficial for a variety of workloads [32][42][31]. In cloud systems a number of studies have confirmed these benefits for parallel and distributed workloads with improvement of performance metrics such as job completion time and power efficiency which correspondingly contribute to reduced cost under the utility computing model [31].

Previous works on the heterogenous cloud (discussed in previous sections such as 3.1.2 and 2) have mainly focused on purely scheduling problems or limited exploiting heterogeneity in the cloud to scenarios such as heterogeneous scaling largely only in the context of different virtualized resource allocations and configurations, such as using large and small sized VMs [31]. This is only a small slice of the true heterogeneity now available in the cloud. We propose one of the causes of this gap is because the format and runtime environment of these virtualized applications makes it difficult for developers and operators to exploit non-architecturally homogeneous scalings with out an overall heterogeneous framework, much as in other types of heterogeneous systems.

Container runtimes treated as purely IaaS can certainly execute containers in a variety of hardware and virtualized contexts. But the ability of container-based frameworks, utilized as PaaS, to exploit heterogeneity through actions such as heterogeneous scaling is limited by the increased complexities that are exposed to the software developer by such scenarios. For example, differing container images are required due to the varied hardware we must exploit and these need to be provided by the application developer. This is unfortunately a consequence of containerized applications fundamentally being composed of binary applications for the operating system being virtualized which are not designed for heterogeneity and this means the resulting container is only really useful on a fairly homogeneous set of cloud systems. In addition, often manual constraints must be manipulated in the orchestration system to achieve a proper scheduling in the presence of heterogeneity, because it is not aware of the specific resources involved [40][39].

We consider this a notable shortcoming as developers turn to container systems to help isolate them from the particulars of the underlying node and software configuration [35]. In order to extend containerized applications to heterogeneous systems without costly and uncloud-like customization by the developer an additional framework is required on top of what is normally provided by container systems. A unified approach and framework utilizing container-based application virtualization in the presence of a potentially heterogeneous cloud environment is required. One which allows the application images to remain independent of any particular system but allows running applications to maximally exploit the particular features of the execution environment they are assigned to.

3.2.1 Bytecode Containers for the Cloud

Incorporating heterogeneous devices such as GPUs as first class citizens in the container-based cloud has begun with works such as those by NVIDIA on Kubernetes [40] and works on heterogeneous scheduling

are becoming more common [14][33][31]. But so far the challenge of mitigating the need to dedicate software development effort to building multiple container images for each supported CPU architecture (and potentially microarchitecture if optimal performance is desired) has not been fully explored.

In order to be able to generate portable, hardware agnostic containers we must ensure the binaries in the container are portable and independent of any particular final target. In order to accomplish this goal we propose utilizing a bytecode format for containerized applications that will allow them to remain independent of the hardware platform but allowing for later robust optimization of the final executable container image. Secondly to accompany this change we propose reforming the container lifecycle to allow for a two-phase compilation/transformation for the container, with responsibility for the final translation to executable format of the container moved from the application developer to the cloud instead, either as part of the Cloud Service Provider (CSP)'s platform or as provided by what we term a separate intermediate Optimization-as-a-Service (OaaS) provider.

3.3 Overview of Our Implementation: Man O' War

Now that we have seen how our goal of ensuring container portability between different cloud systems that are architecturally heterogeneous can be met with a bytecode-base approach we can discuss the specifics of our implementation. In this section we describe the Man O' War system which currently supports the C programming language (though we hope to see it extended to other statically typed languages in the future) and focuses on portability between CPU architectures that are common in cloud data centres (specifically x86 and ARM). We focus our implementation on solely CPU architectures to validate the technique but we discuss how this approach maybe extended to incorporate the work that has already been done with parallel accelerators in our final chapter.

In order to be able to generate truly portable containers, we must ensure the binaries in the container are portable and independent of any particular final target. We realize our bytecode container approach using a portable LLVM bitcode-based IR for the application(s) that are stored in the container. The resulting standard container image can still be stored and handled using OCI compliant tools.

A container "finalizer", using the terminology derived from Heterogeneous System Architecture (HSA)[42], is added to the platform which uses an LLVM-derived translator to processes the container images into their final natively executable format. The design of this translator is shown in the top-right of Figure 3.2. We must make some extensions to the OCI container format to allow it to be processed in such a way. Specifically, any platform binary code in the container must be marked in a special manifest so the translator can efficiently process the image.

We consider two primary ways for the finalization process to be integrated into the lifecycle of an existing container system. Either as an external service, which we term OaaS, or embedded within the container-based PaaS. In either cased when knowledge of the target platform becomes available, either in advance or on demand once a container instance is requested, the finalizer runs the translator over the various binaries in the image converting the LLVM bytecode to the native instructions of the target architecture/microarchitecture. The result of this is a new container image optimized for that specific

target.

Using the Man O' War toolchain, developers of applications for the platform no longer build their application containers for a generic version of their preferred hardware architecture (i.e. x86_64-pc-linux), but instead for our new virtual platform (i.e. le32-manowar-linux). The toolchain provides a Portable Operating System Interface (POSIX)-like split C standard library that complies to the newly defined Man O' War virtual Application Binary Interface (ABI) for our le32 virtual architecture. This split standard library functions as an abstraction layer, serving to isolate the application from some of the difference between Linux ABIs on different hardware architectures. Other than utilizing the new toolchain, their container build process can remain very similar to what it was for targeting an ordinary hardware platform. The resulting application is stored as LLVM-based bitcode in the container image.

3.3.1 Components

We provide the toolchain, container finalizer and other necessary tooling to process the container image and integrate it with OCI container runtimes. This implementation includes modified open source components from Portable Native Client (pNaCl)[6], LLVM [18], Musl [21] and a variety of other projects.

The main components of the Man O' War systems are as follows:

- toolchain - the compiler frontend, linker and other tooling that takes a C application and translates it to LLVM-based bitcode, along with various scripts to integrate this with the container build process
- libraries - the split libC implementation that presents a portable partially POSIX compliant API conforming to our le32-manowar-linux intermediate platform and the backend platform library used by the translator that adapts it to the conform to the native ABI of the target platform
- translator - the translator frontend that takes the bytecode image, performs optimization and generates machine code from it and preforms linking to the platform libraries
- container finalizer - a tool that take a description of the target node hardware and the portable container to be translated and runs the translator over all the binaries in the container manifest, creating an optimized platform container

3.4 Man O' War Virtual Target / ABI

We define a target virtual ABI (i.e. le32-manowar-linux) which defines a 32-bit little endian target with a ILP32 data model [12]. This is necessary as the C programming language standard intentionally leaves a great number of things, such as the size and alignment of types up to the specific platform implementation and ABI[13].

It is also important to note that LLVM bitcode is not inherently portable. For example the C frontend, Clang will embed a great deal of platform/target specific information into our bitcode that originates both from platform intrinsic within Clang and target specific headers / libraries. For Clang

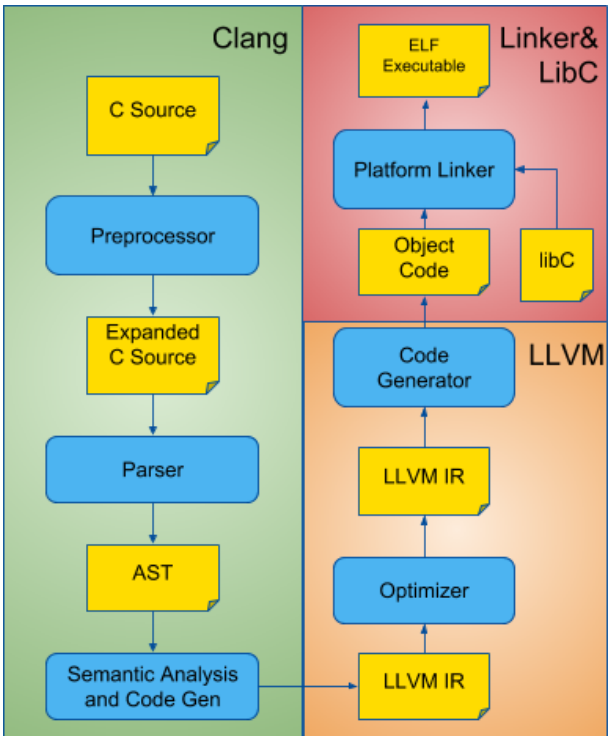
to generate consistent, portable LLVM IR it must be generating bitcode for a virtual target platform that has a consistent ABI that conforms with our expected behaviour for the bitcode, such as calling conventions and other parameters. Thus the need for this meta-platform which will ensure we generate consistent, portable bitcode. Details for the actual platform ABI will be provided later by the translator.

We do not attempt to list a full platform definition here, but beyond the fact that it uses a 32-bit little endian with an IPL32-type datalayout we note that floating point limited to the IEEE 754 standard (floating point support overall is currently quite limited) and relies on LLVM atomics [6]. These limitations are sufficient for demonstrating the sample workloads included in our evaluation chapter and are in fact sufficient for a great many real world applications as well, as adoption of the new x32 ABI on amd64 platforms suggests [30].

3.5 Container Toolchain

Our goal was to produce a largely POSIX-compatible toolchain to allow typical Linux Standard Base (LSB) compatible containerized applications to be recompiled for our new container architecture with no or minimal modifications using standard container build tools. We can see the flow of a typical Clang/LLVM toolchain in Figure 3.1.

Figure 3.1: A standard Clang+LLVM C compilation on Linux

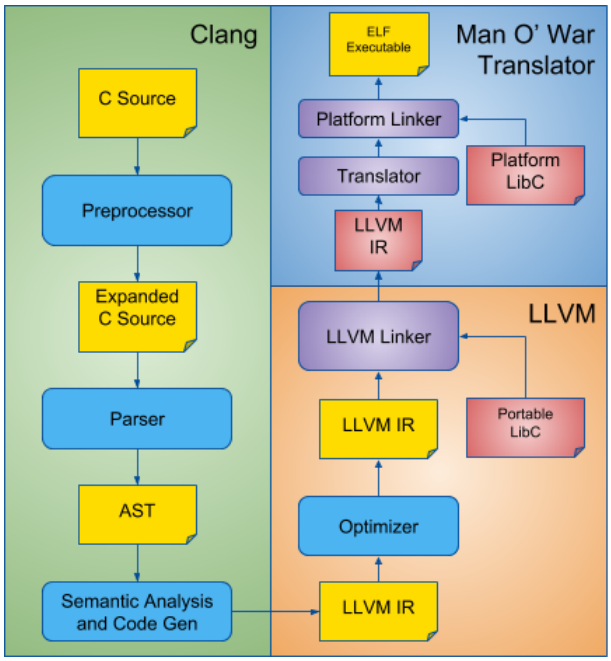


As a starting point we utilized the Google Native Client code bases implementation of the pNaCL

LLVM/Clang toolchain. This implementation is composed of a python-based compiler driver script and frontend for the toolchain, a customized version of LLVM and Clang, as well as a translator to convert the bytecode version of of the executable to its final platform specific form, as well as a set of libraries which called Pepper which attempt to implement POSIX APIs on top of the Chrome Browser environment.

From this starting point we have modified the included LLVM/Clang removing unnecessary optimization, sanitation and simplification steps target at step required for their application software fault isolation (SFI) sandboxing of the *Pepper* APIs and the Chrome environment. We have added our own heavily customized split version of the Musl standard library, implementing the standard library-as-abstration layer from Linux ABI differences. As well we add a versions of GNU binutils and the LLVM project compiler-rt (which provides necessary compiler runtime intrinsics) as platform libraries for use by the translator. A diagram of the new toolchain flow with the translator is shown in Figure 3.2.

Figure 3.2: Man O' War Toolchain



Our implementation is not yet feature complete to a large amount of functions of a POSIX environment, with limited support of advanced floating point, threading and atomics libraries, as well as dynamic linking, as these were unnecessary for the validation of our design and the performance experiments that follow later in this work. However we have achieved our goal in that any number of standard applications not requiring these features, for example GNU utilities, SPEC benchmarks and other applications without complex external dependencies, can successfully be ported and run with the toolchain.

3.6 Split Standard Library

One of the difficulties we encounter with applying this technique to a modern operating system platform is that the Linux ABI is not entirely consistent across platforms. If we look at the Linux Standard Base specification we see that each hardware platform has it's own addendum [8] giving platform specific differences. This is a necessity as, for example, the method to invoke system calls is by it's very nature platform dependant.

We also find certain data structures which are exposed to user space differ in their implementation by hardware platform. For example, the ucontext_t structure, used as part of the sigaction system call to setup a signal handler, has many fields who's bit width and alignment depend on that of the host architecture. We demonstrated an example of this in the Background chapter section on LLVM in Listing 4. Since these definitions are normally picked up from the system headers when a Linux program is compiled, and thus would be included in any resulting bytecode, we must address this issue if we wish to have a portable bytecode implementation.

Our implementation of the C standard library is based upon a fork of the open source Musl C library [21], but has been drastically reshaped into two separate components (see Figure 3.3). Firstly into a portable libC library interface conforming to our *le32-manowar-linux* virtual platform that can be linked at bytecode generation time and secondly a platform library containing all assembly and platform specific routines that is linked in at translation time. The platform library the acts as an hardware abstraction layer, converting any non-portable kernel types into portable equivalents before exposing them to the application code.

Generally this can be done by making their field ordering and sizing consistent, padding if need be. It is also necessary to replace certain preprocessor definitions (i.e. #define) which are platform specific with global constants which can be resolved during linking (see an example for signal numbers in Listing 1). While this makes the resulting library not precisely POSIX compliant, we have so far found that most well written applications require little to no modification to be compiled.

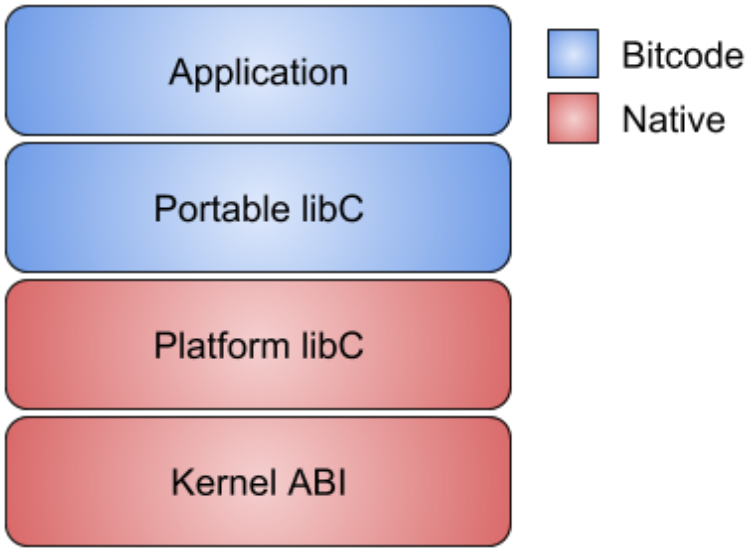
```
// musl
#define SIGHUP    1

// manowar portable
#define SIGHUP SIGHUP_g
const int SIGHUP_g;

// manowar platform
const int SIGHUP_g=    1;
```

Listing 1: Ucontext_t as defined by the Musl C Standard Library

Figure 3.3: The split C library acts as an abstraction layer for the Kernel ABI differences



3.7 Man O' War Image Format

In this section we define the format of the container so it can be processed by the finalizer. The format for Man O' War container images is defined as an extension to the OCI image format and we use field names and attributes that have the meaning specified in that standard. Specifically the value of certain fields and attributes is extended to allow the image to be processed by the finalizer tool. What follows is a brief description of OCI image format, these extensions and then an explanation of how the image is process by the finalizer tool.

OCI Image Format

The OCI Image Format provides a standard format for containers, allowing them to be processed and run by a variety of tools, including our own. The image is fundamentally composed of a number of layers which represent delta changes composing additions, modifications and deletions to a previous layer. It is composed of the following primary components [25]:

- Image Manifest - a document describing the components that make up a container image
- Image Index - an annotated index of image manifests
- Filesystem Layer - a changeset that describes a container's filesystem
- Image Configuration - a document determining layer ordering and configuration of the image suitable for translation into a runtime bundle

An index may point to many different manifests for the same image, containing for example versions of the image for different hardware architectures. The manifest in turn points to a specific set of filesystem layers and a configuration for this instance of the image.

Description

In the Man O' War image prior to processing by the finalizer, the **Image Index** must contain an entry with the value of **architecture** set to the value of *le32-manowar* and **os** set to *linux*. The corresponding **Image Configuration** must similarly have **architecture** set to the value of *le32-manowar* and **os** set to *linux*. This indicates the bytecode images manifest and filesystem layers.

In the resulting image created by the finalizer after processing, the **Image Index** will contain an entry with the value of **architecture** set to the value passed to the finalizer along with optionally the cpu architecture **variant** (eg. armv7) and optionally the following attributes show in Table 3.1.

Attribute	Value
space.manowar.cpu	a string value indicating the CPU type in the format specified by LLVM (eg. skylake)
space.manowar.features	a string value containing a comma delimited list of the CPU features enabled for this build

Table 3.1: Optional attributes for post-processing Man O' War images

3.8 Finalization

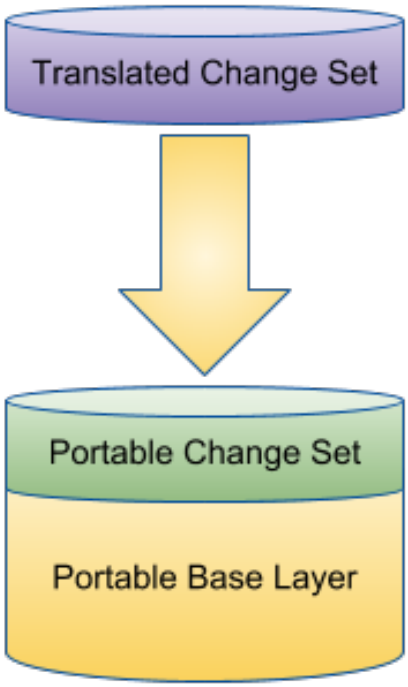
In order to convert the Man O' War bytecode image to a regular OCI-format container containing standard platform executables so it can be run by an unmodified container engine, we process it with the finalizer tool. This tool apply the translator to all bytecode binaries in the image to generate standard platform binaries. The Man O' War on-disk image format is laid out as specified in the OCI image specification.

3.8.1 Operation

The finalizer expects to find a file called *.MANOWAR-MANIFEST* in the root of the filesystem once all changesets are applied. This manifest contains a null-delimited list of the bytecode files in the image that require translation. It runs the Man O' War translator over each file with the target architecture, cpu and feature flags defined by the cpuinfo.json file passed to it.

The translator itself consists of tool which invokes a version of the LLVM bitcode to assembly translator (llc) which performs optimization and code generation along with a linker to pull in the platform and runtime libraries required.

Figure 3.4: The finalizer creates a new layer containing the translated binaries



This results in a changeset replacing each bytecode file with it's translated native version, which is added to the original filesystem layers as show in Figure 3.4. This changeset is used as the root filesystem for a new manifest in the container image index and the configuration is duplicated and then updated as specified above.

The CPU info JSON blob accept by the finalizer is formatted according to Table 3.2. While some of these properties are optional, they are all heavily recommended. A tool, *procinfo*, is provided to extract this information.

Attribute	Presence	Value
architecture	mandatory	the target architecture in the standard OCI format
variant	optional	the variant of the architecture in the standard OCI format (eg. armv7)
cpu	optional	a string value indicating the CPU type in the format specified by LLVM (eg. skylake)
features	optional	a string value containing a comma delimited list of the CPU features enabled for this build (eg. sse4.1)

Table 3.2: CPU Info JSON Properties

It is important to note that the resulting images are intended to be heavily cachable to offset any extra cost of running the extra finalization step. Container runtimes utilizing the Man O' War system and

finalizer can utilized the attributes encode in the the finalization step to see if an acceptable translation already exists for the particular architecture, cpu and feature flag combination. The translated version of these images can also be stored in whatever image repository the particular orchestration engine in use is utilizing, as the translated images are fully OCI compliant, there by making them available to the whole container system.

3.9 Integration with Existing Container Architectures

The Man O' War container format and toolchain are designed to be flexible enough to be incorporated in a variety of container-based systems and orchestration technologies. In this section we will discuss integration strategies with existing container architectures and describe how we support them.

The specific way that portable bytecode container systems will be integrated will depend on the end users desired mode of use and control over the underlying cloud platform, but we envision two primary modes of integration. The first scenario is that the cloud platform will have native support for bytecode containers, with the platform specifically managing building and deploying native containers from the externally supplied bytecode container image. This is applicable in either PaaS where the container service integrates this capability or in an on-premise scenario where modifying and customizing the container runtime is possible.

The flow of events for a user of such a service (assuming no cached image) would be something similar to what is show in listing 2:

1. Request is sent to PaaS to run bytecode image
2. PaaS fetches bytecode image from repository
3. PaaS runs a separate finalizer process over the image using its internal node configuration
4. PaaS runs the resulting image

Listing 2: Bytecode-aware Container-based PaaS Workflow

Alternatively, in the case the container platform is not natively bytecode aware an external service, what we term a OaaS provider, translates and optimizes images either integrated with or acting as a container registry. The translated images can then be run as normal on unmodified container runtimes.

The flow of events for a user of such a service (assuming no cached image) would be something similar to what is show in listing 3:

3.9.1 Integration Example with Kubernetes

We will use native integration with Kubernetes (k8s) as an illustrative example of how a bytecode containers could be integrated with existing container systems, as it is open source and by far one of

1. Request is sent to OaaS provider to finalize image along with node configuration
2. OaaS provider fetches image from repository and finalizes it
3. OaaS uploads finalized image to repository
4. Request is sent to PaaS to run finalized image

Listing 3: Container-based PaaS and OaaS Workflow

the most popular orchestrators today, with deployments in both Google Cloud Platform (GCP) (where it was originally developed) and in Microsoft Azure, among other public clouds.

Kubernetes' support for alternate container engines is in ongoing development, CRI-O is a plugin attempting to make it easy to incorporate Open Container Initiative (OCI) compliant runtimes into Kubernetes but it is still an incubating project and has not yet achieved full acceptance. Accordingly we will only remark on the high level attributes of such an integration.

Kubernetes consists of a control plane, consisting of one or more masters running the API Server and Controller among other components in containers, and the various nodes systems that actually run the workload. The node systems run a daemon called the kubelet, which accepts commands from the master and communicates with the local container runtime to create and take containers scheduled to the node through their life cycle.

In order for the container images to be finalized we will need to add a custom service to the control plane that is responsible for building the images when required. When the container runtime is commanded to run a container via kublet that contains a manowar image in it's manifest, it contacts the service to request a new image be built or a cached one be sent. As finalized Man O' War containers are OCI compliant, once a finalize container arrives from the finalization service the image will run as normal.

In this setup a builder node can be run by the service which fetches the container image as well as the nodes CPU info JSON blob via an http service running on the node. It then finalizes the various container images and caches them in the registry for future use. This can be rendered very storage efficient due to the fact the containers are implemented as change-layers and the additional binaries will not likely require significant storage.

We have discussed how bytecode techniques may help us to bridge the gap between container-based applications and exploiting the potential of the heterogeneous cloud. Together the virtual ABI, toolchain and translator, container finalizer, and novel split C standard library, and miscellaneous tools together form the components of the Man O' War system. Together they allow us to compile a standard, POSIX-compliant C application into a portable bytecode containerized application. The resulting container can be translated on demand using detailed information about the node/system it will be run on, enabling localized architectural/micro-architectural optimizations for it's placement in the heterogeneous cloud.

We will demonstrate via our experiments in the next chapter that this implementation can achieve notable performance benefits in a variety of scenarios in the heterogeneous cloud, such as heterogeneous scaling between architecturally dissimilar nodes. There by we hope to further demonstrate the utility of bytecode for use with containerized applications in heterogeneous environments. We will also focus on demonstrating the larger scale optimization/performance potential of portable containers in the heterogeneous cloud, demonstrating we can use bytecode to optimize in a scalable way and there by achieve notable improvements in overall resource efficiency.

Chapter 4

Evaluation and Validation

In order to demonstrate the advantages of a bytecode-based container framework and format in the heterogeneous cloud we will evaluate our implementation upon a number of benchmarks. The first experiment deals with that of a heterogeneous scaling scenario involving migrating between architectural dissimilar node types within the cloud. The second compares the performance of a standard container built in a traditional fashion with bytecode containers and looks at the impact of any overhead of the runtime itself and of optimizing locally based on the architecture/microarchitecture available on a node.

We rely on the standard Standard Performance Evaluation Corporation (SPEC) CPU 2017 performance testing benchmarks to compare performance [36]. These benchmarks "focuses on compute intensive performance" [36] and different versions of them are used for evaluation by various bytecode works including Lattner *et al.*[18], Yee *et al.*[45], and Nuzman *et al.*[24], which are discussed in our background chapter. Specifically we use variants of the LBM benchmark from the floating point family which "implements the so-called 'Lattice Boltzmann Method' (LBM) to simulate incompressible fluids in 3D"[36] according to the SPEC documentation, which implements a fairly typical scientific workload. Our implementation is limited by design to the C programming language, so only those benchmarks written exclusively in C are available to be considered and as our current implementation does not implement a complete set of POSIX functionality as of yet a selection of these benchmarks are also not available, restricting our choices further within the available SPEC benchmarks.

By running them in a containerized environment with both a tradition build with fix optimization settings and a bytecode build, which allows for flexible optimization to the heterogeneous environment, we will be able to evaluate potential performance gains made feasible by adopting bytecode containers as well as potential overheads of our technique. All benchmark trials are setup and run on the same nodes in the same configuration to minimise and control for effects from variables such as latency to attached storage and other system behaviours that may vary from instance to instance in the cloud. As with all benchmarking, specific measurement and improvements depend on a particular workload and configuration so our goal is not to demonstrate a specific numerical improvement but rather demonstrate the utility of the method and system in that type of scenario.

4.1 Evaluation on a Heterogeneous Scaling Scenario between CPU Architectures

4.1.1 Purpose

Using a repeated test with a synthetic workload, we will compare the performance of a bytecode version of an application built with our toolchain which is moved from a high power x86 node to a low power ARM node. This corresponds roughly to the heterogeneous scaling case we considered in our prior discussions. While we expect a notable degradation in overall execution time due to the use of a low-power core, we use this data to compute the expected difference in execution costs, a key practical metric from public cloud systems. This presents a great opportunity for heterogeneous scaling and optimization as it allows us to dynamically migrate nodes based on their changing performance vs cost requirements, moving deadline insensitive and non-interactive tasks to where they are cheapest to perform based on the current (and possibly dynamically changing) cloud pricing.

4.1.2 Experimental Setup

We rely on the standard Standard Performance Evaluation Corporation (SPEC) CPU2017 performance testing benchmark to compare performance, specifically the 519.lbm_r floating performance benchmark was run. This variant of the benchmark is a completion rate based benchmark measuring the work completed per unit time. As a rate based experiment it can represent a set of request to a computationally intensive cloud-based service, for example a number of requests to a video processing web service, or a set of long-running jobs such as in transaction processing. The rate-based benchmark also has significantly reduced peak memory requirements however which is important for our memory constrained ARM test devices.

On x86 we utilized a Digital Ocean performance instance in a 4 virtual core configuration with 8 GB Random Access Memory (RAM) and a Intel(R) Xeon(R) CPU E5-2697A processor. On ARM a Scaleway C1-type dedicated node which is a 4-core Marvel Armada ARMv7 with 2 GB RAM was used. The test operating system was Ubuntu 16.04 LTS in both cases.

4.1.3 Methodology

The image was built with with the Man O' War bytecode toolchain and translator on an x86-based Digital Ocean public cloud instance while providing the toolchain information about the micro-architecture and available processor extensions. The SPEC tools were the utilized to run the benchmark 9 times with a workload size of 4 copies (1 per core) and gather the performance metrics which are recorded bellow. The process was repeated identically again on the ARM-based Scaleway public cloud instance with the exception that the benchmark was repeated only 3 times owing to the significantly longer time it takes to run on ARM[1].

[1]Total execution time for ARM was more than 8 hours.

4.1.4 Data

Table 4.2 and 4.1 on page 37 contain the results of our experiments. Note again that fewer runs were able to be preformed on ARM as a result of the longer total execution time. A graphical comparison is shown in Figure 4.1.

Table 4.1: SPEC CPU 2017 FPRate 519.lbm_r with Man O' War on ARM

Benchmarks	Copies	Estimated Run Time
519.lbm_r	4	9823
519.lbm_r	4	9611
519.lbm_r	4	9590

Table 4.2: SPEC CPU 2017 FPRate 519.lbm_r with Man O' War on x86

Benchmarks	Copies	Estimated Run Time
519.lbm_r	4	900
519.lbm_r	4	880
519.lbm_r	4	853
519.lbm_r	4	742
519.lbm_r	4	736
519.lbm_r	4	1512
519.lbm_r	4	1007
519.lbm_r	4	850
519.lbm_r	4	894

4.1.5 Analysis

The mean time for the workload on lower power ARM was 9675s with standard deviation of 128.9s which with a 95% confidence interval gives us a runtime of $9675 \pm 145.9s$. The mean time on high power x86 was 930 with standard deviation of 233.1s which with a 95% confidence interval gives us a runtime of $930 \pm 152.3s$. This means the overall work per unit time performance drops to an average of 9.62% of what was available on the x86 version. However, with a cost of €0.006 per hour (approximately US $0.007 as of August 3rd) for the ARM node vs a cost of US $0.119 per hour the total cost of the job on average is 63.41% more on x86. We compute the cost-delay product (CDP) as proposed by Roloff *et al.*[31], as show in equation 4.1, below as a comparison metric:

$$CDP = cost\ of\ execution\ \times\ execution\ time \tag{4.1}$$

$$CDP_{x86\_instance} = 0.119 \times 930$$
$$= 110.67$$

$$CDP_{arm\_instance} = 0.007 \times 9675$$
$$= 67.725$$

A comparison of the CDP metric can be seen in Figure 4.2 (lower is better). While the exact ratio is dependant on provider pricing and other external factors, we argue that this is is no mere coincidence. Fundamentally we can consider these processor types as points in the engineering design space optimized for different levels of performance [3]. We note that the Intel Xeon(R) E5 processor family is designed as a high-performance server process with a thermal design power of 145 W on the E5-2697A. Meanwhile the Marvel Armada is a much lower power System on Chip (SoC) with a maximum power dissipation of 1.650W. These power metrics do not capture all the power usage of the system or on our particular workload (this would be difficult to measure accurately on public cloud), but they give us a general idea of the relative power consumption which we compute below:

$$E_{x86\_instance} = 145W \times 930s$$
$$\approx 135kJ$$

$$E_{arm\_instance} = 1.65 \times 9675s$$
$$\approx 15.9kJ$$

Given that power utilization is an important cost in modern data centres, it is not surprising that these lower power processor designs are available at a substantially reduced cost (in addition the System-on-Chip (SOC) design is also inherently lower cost to produce). Thus we believe there are significant cost savings that can be achieved by using bytecode systems such as Man O' War to exploit the diverse architectural/microarchitecural nature of heterogeneous clouds based on the particular needs of the workload.

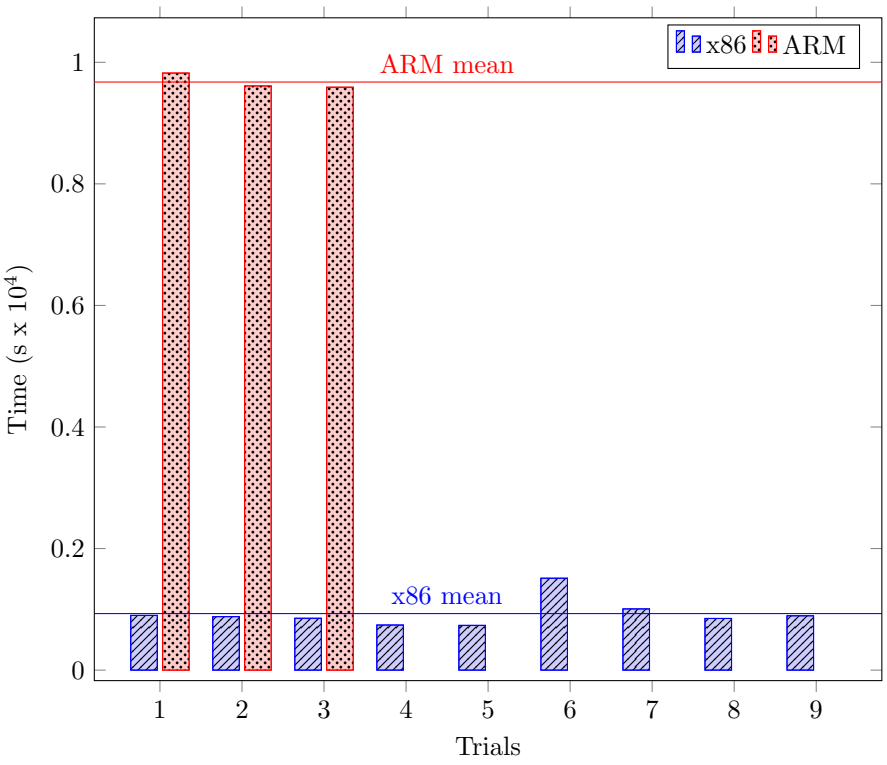


Figure 4.1: 519.lbm_r bytecode Execution Time on ARM and x86

Figure 4.2: 519.lbm_r bytecode Cost Delay Product (CDP)

4.2 Performance Evaluation between Traditional and Bytecode Image

4.2.1 Purpose

This scenario explores the performance characteristics of bytecode vs traditionally compiled containerized applications. In order to be useful in reducing the software development effort for heterogeneous systems in the cloud, software developers should have confidence that any overheads introduced by the technique will not results in insurmountable performance costs. Indeed, one of the goals of our approach is to produce binary containers from bytecode that are utilized and run in exactly the the same manner as their traditional predecessors.

We will run the workload with a container build in the standard fashion with the default optimizations for the platform. We will then run the workload with a bytecode container finalized with specific information about the platform microarchitecture. It is a common scenario that newer cloud servers underlying the container platform will have available new processor extensions and microarchitectures, such as for example the Advanced Vector Extensions 2 (AVX2) instructions available starting in Intel's Haswell, that may well also for additional optimization. This will let us demonstrate the performance impact of adopting bytecode-based containers, giving an insight to any overheads of this method while potentially gaining any benefits from the microarchitectural information available. Again we note that this evaluation is far from exhaustive given the current preliminary state of our implementation, but it will serve to illustrate what is possible with familiar workloads using bytecode containers.

4.2.2 Experimental Setup

In this experiment, using a repeated test with a synthetic workload from SPEC CPU2017, we will compare the performance of natively compiled and bytecode versions of applications built with our toolchain. We have two goals with this experiment, the first primary goal is to evaluate the overhead of our split library implementation and other components of our system and to demonstrate the lack of any potential negative performance impact. The second goal is to see whether passing the processor information including details on the microarchitecture and instruction set instructions results in any notable performance improvements or better use of available processor features.

The 619.lbm_s floating performance benchmark was select to be run as our performance test. Our implementation is limited by design to the C programming language, so only those benchmarks written exclusively in C are available considered and as our current implementation does not implement a complete set of POSIX functionality as of yet a selection of these benchmarks benchmarks are also not available restricting our choices further. The 619.lbm_s benchmark is a speed based benchmark from the floating point family. We think this benchmark is suitable and of interest as it is representative of demanding analytic and scientific tasks and it could potentially benefit from advanced floating point features such as AVX/AVX2/AVX-512 that are available on newer processors. We choose the speed variant because we wish to extract detailed performance profiling for a single run of the benchmark with

minimal interference from competing processes.

4.2.3 Methodology

The benchmark was run in a sixteen thread configuration, with the score (expressed as a ratio to SPEC's reference machine) and completion times being the lowest of the three. It was run first with Clang, LLVM, MuSL and CompilerRT version which are included in or exactly match the ones in our toolchain, so the performance scores rely on only the available optimizations. We then ran it with the Man O' War toolchain and translator, providing the platform information about the microarchitecture and available processor extensions. The test system was a DigitalOcean Performance cloud instance with 8 virtual cores and 16GB running an Intel(R) Xeon(R) Platinum 8168 CPU processor and Ubuntu 16.04 with Docker CE.

4.2.4 Data

Table 4.3 and 4.4 on page 42 contain the results of our experiments. A graphical comparison of the same is shown in Figure 4.3.

Table 4.3: SPEC CPU 2017 FPSpeed 619.lbm_s with LLVM+Clang+Musl+CompilerRT on DigitalOcean

Benchmarks	Threads	Estimated Run Time (s)	Ratio
619.lbm_s	16	1808	2.897
619.lbm_s	16	1820	2.878
619.lbm_s	16	1816	2.884
619.lbm_s	16	1839	2.848
619.lbm_s	16	1832	2.859

Table 4.4: SPEC CPU 2017 FPSpeed 619.lbm_s with Man O' War

Benchmarks	Threads	Estimated Run Time (s)	Ratio
619.lbm_s	16	1807	2.898
619.lbm_s	16	1752	2.989
619.lbm_s	16	1819	2.879
619.lbm_s	16	1822	2.874
619.lbm_s	16	1823	2.874

4.2.5 Analysis

The mean for our standard toolchain result was 1823s with a standard deviation of 12.59s and the bytecode mean was 1805s with a standard deviation of 34.61s which with 5 trials and a 95% confidence

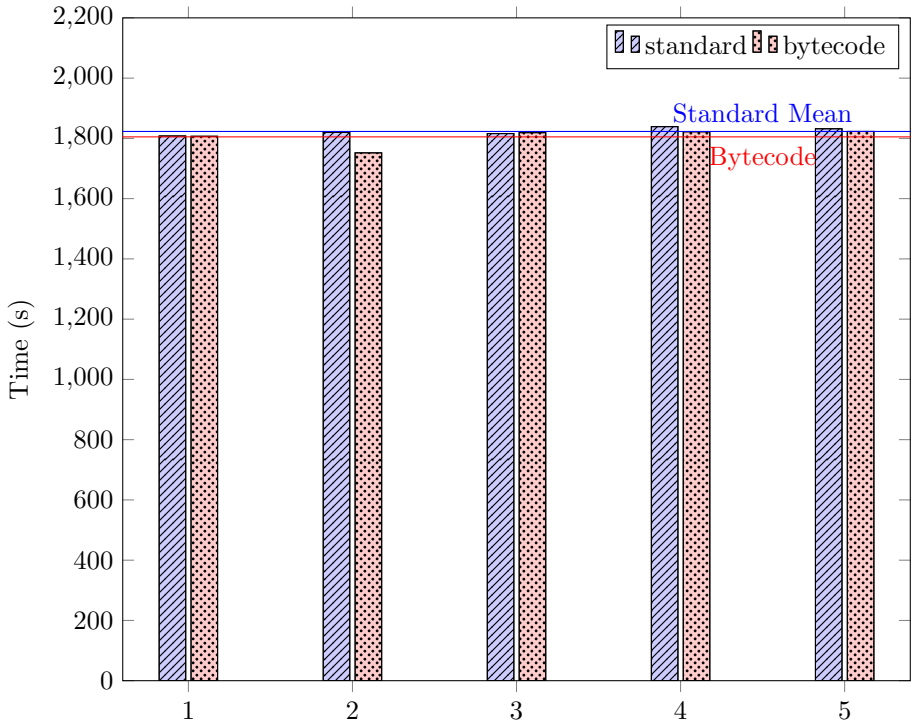


Figure 4.3: Standard vs Bytecode Performance on x86

interval gives us an average runtime of $1823 \pm 11.04s$ and 1805 ± 30.34 respectively. In Figures 4.4, 4.5 and 4.6 starting on page 45 we can see flame graphs based on measured on CPU-time per function measured by profiling the two executions using the Linux Perf tool [10]. A enlarged section of the graph is also shown, indicated just the tops of stacks descending from the main sections of the program. By comparing these graphs (and the original Perf data) we can see that the programs behaviour and portion of CPU time spent executing various functions are largely similar, excepting variable system behaviours such as page faults. LBM_performStreamCollideTRT, which is forms a key part of the numerical simulation, is on CPU in 99.57% of samples in the traditional case and is on CPU in 99.56% percent of samples in the bytecode case.

Combined with the statistically similar runtime from the SPEC test cases we can assure ourselves that the execution behaviour of the bytecode version is correct and largely similar and we can conclude that the overall impact of the bytecode translation and the any overheads added by the split C library and abstraction layer appears to be very minimal. Even for a program which is I/O-bound rather than CPU-bound we do not expect a significant change as our implementation does not in fact necessitate adding additional abstraction-levels to I/O functions, as these are largely independent anyway [8].

Dissasembly of the resulting executable does in fact show the presence of AVX/AVX2 extensions[2] not used by the original container. However they are present in a limited fashion that likely had

[2]When the toolchain was allowed to generate AVX-512 instructions, the compilation failed due to what we suspect are issues generating these instructions in an x32 environment.

no measurable impact, and thus passing the detailed architectural information appears to have had a limited overall performance impact in this case. While this workload does have the appropriate parallelism under-utilization of available SIMD instructions can occur if the LLVM optimizers are not able to able to effectively vectorize the operations available and our choice of newer optimizers is limited by the version of LLVM we utilize[3]. Overall this shows that automated microarchitectural optimizations are indeed possible with our bytecode container approach, though their benefits will depend upon the available translator.

In total, these results are very promising because it means bytecode containers can potentially be used in place of regular containers even in homogeneous environments with no notable overheads. This enables their use without concern of performance regressions and enables their use for their many benefits to the software development process in the presence of heterogeneity. Also there remains the possibility of optimizations based on the microarchitecture depending on the workload and tooling.

[3]The older version of LLVM used is caused by the dependencies of various components within our toolchain.

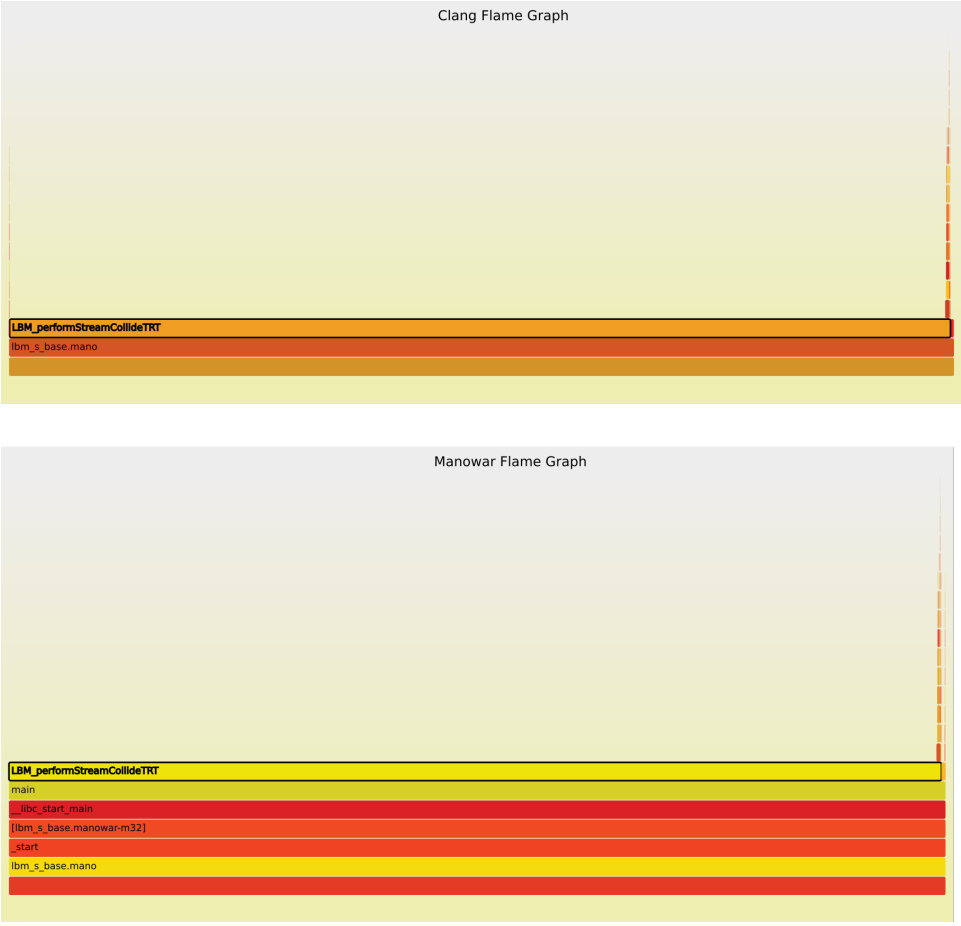


Figure 4.4: 619.lbm_s Flame Graphs. The length of a box along the horizontal axis shows the portion of time spent on CPU out of all samples. The vertical axis shows the call stack. Both calls to LBM_performStream-CollideTRT in the traditional Clang and Man O' War version dominate the execution and run in the same proportion of total samples. Combined with the SPEC data this leads us to believe both versions are performing similarly.

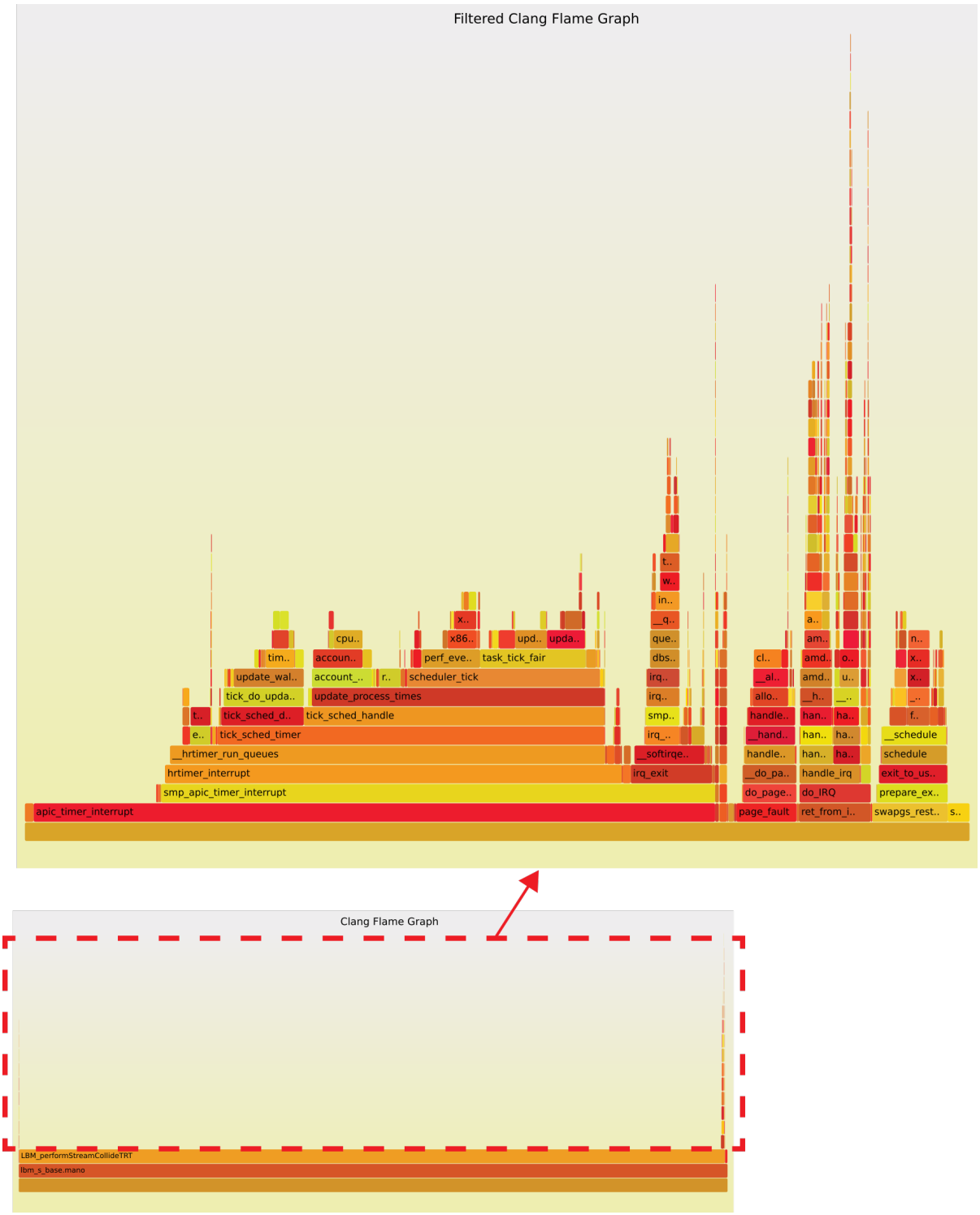


Figure 4.5: 619.lbm_s Standard Toolchain Flame Graph but filtered to highlight stack tops by dropping last several common frames. The length of a box along the horizontal axis shows the portion of time spent on CPU out of all samples. The vertical axis shows the call stack.

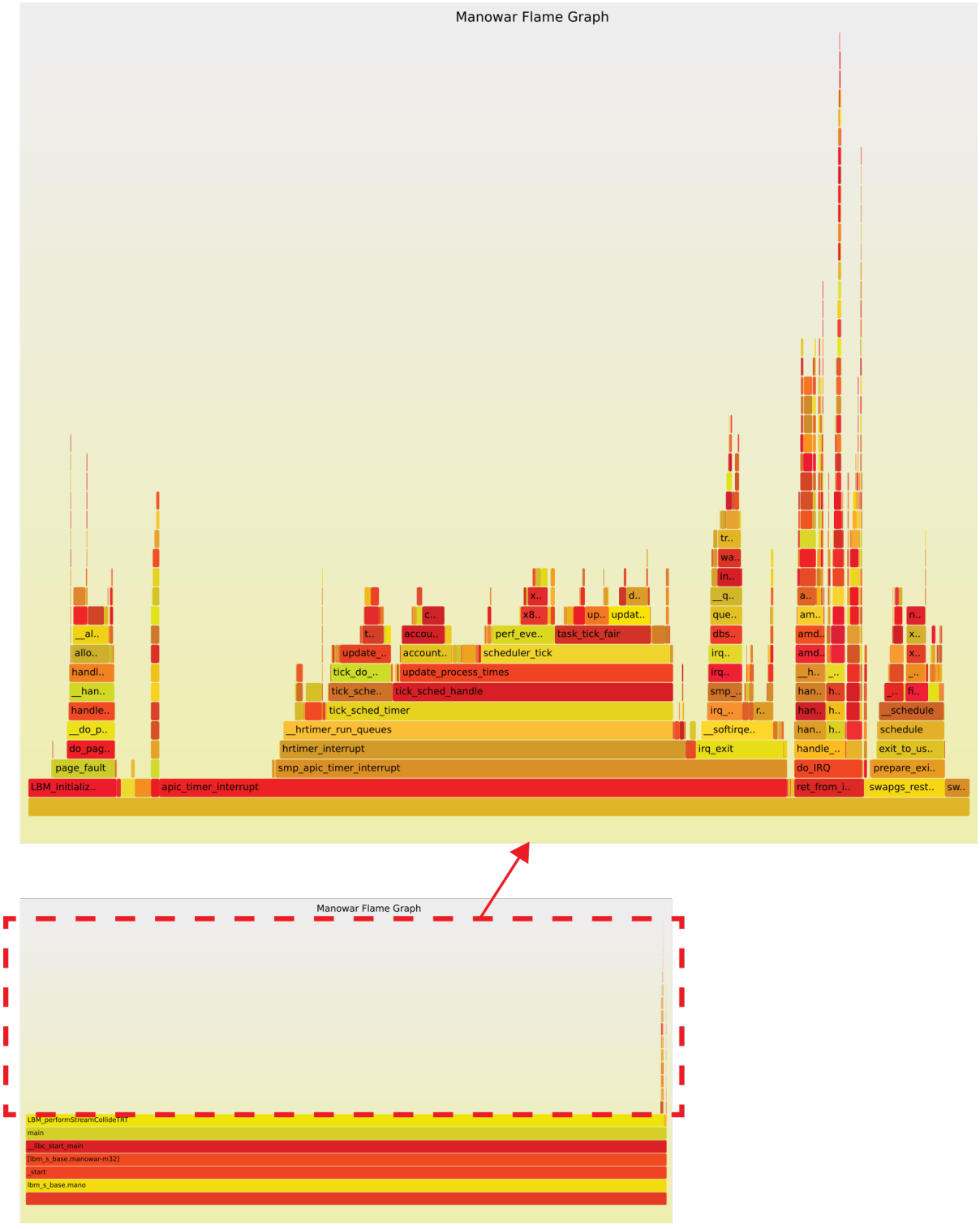


Figure 4.6: 619.lbm_s Manowar Toolchain Flame Graph but filtered to highlight stack tops by dropping last several common frames. The length of a box along the horizontal axis shows the portion of time spent on CPU out of all samples. The vertical axis shows the call stack.

4.3 Discussions and Comparison with Related Work

In this section we make comparison with and discuss related work introduced in our background, reflecting on what separates our work and it's place in the larger context. In this section we give a brief discussion of some of the design decisions made in our approach and end with a comparison to the related work we discussed in this section.

We begin by a comparison with existing heterogeneous systems frameworks, discussing what sets our technique apart from simply using these systems in the cloud. We discuss the Heterogeneous System Architecture, a new platform for unifying heterogeneous accelerators using IR and shared memory techniques, which uses promising bytecode-based techniques to solve heterogeneous systems problems and can be used in a containerized setting.

We then discuss several existing bytecode systems related to our work. We begin by discussing LLVM [18], as it forms a key component of not only our work, but several of the other works discussed here and the modern computing landscape as a whole. Next we discuss the paper "JIT Technology with C/C++: Feedback-Directed Dynamic Recompilation for Statically Compiled Languages"[24] which discusses many of the limitations with current compiler infrastructure which apply equally well to the heterogeneous cloud and their incorporation of bytecode techniques has strong parallels to parts of our approach. We then move on to discuss Google Native Client [44], which attempts to target multiple client-side computing platforms in the web environment using a bytecode-based technique and which forms a large part of the basis of our toolchain and approach.

4.3.1 Heterogeneous Systems Frameworks

Heterogeneous System Architecture (HSA)

Heterogeneous System Architecture (HSA), discussed in section 2.1.3, is a very promising platform for supporting parallel accelerators and other types of directly connected, shared memory heterogeneous architectures and we consider it complementary to our techniques. We see systems such as Heterogeneous System Architecture (HSA) as a complementary system providing a unified interface for accessing parallel accelerators via the HSAIL IR. We utilize concepts from their architecture such as the online / offline finalization and they do much to advance the concept of mixed-architecture heterogeneous systems. It is also another excellent example of successful bytecode-based techniques for architectural portability.

What sets our container-based heterogenous systems framework for cloud apart from this types of frameworks is that the focus almost exclusively on parallelization and parallel accelerators. They still require a traditionally compiled host program to drive the heterogeneous portion of the computations (they mandate a master-worker architecture) and an consider only single node heterogeneity (i.e. it requires a physical shared memory model between the host and the executing device).

It is certainly possible to use HSA from with containerized applications and integration of HSA into the Man O' War toolchain is an interesting future direction. In our future work section we will discuss possibilities of incorporating it with our approach. Seeing if such a model can be generalised to distributed shared memory systems or similar constructions in the cloud is also an interesting question,

but one that currently remains out of scope for our own work.

4.3.2 Bytecode Systems

LLVM

LLVM and the LLVM bitcode format, discussed in section 2.2.1, forms a core component of our system and indeed modern computing as Adve, Latner and Cheng won the 2012 ACM System Software Award for their contributions and Clang[4]/LLVM is the default compiler for Apple systems and many others. LLVM forms the basis of our approach and provides key optimization techniques such as FDO but in comparison to the work presented here does not by itself allow for portability or easy integration into existing container systems.

Despite clearly separating the front and backends in the LLVM system, there are some limitations due to the nature of the types of languages LLVM works with (primarily C-like ones). C is not a very hardware agnostic language, the C Specification [13] leaves a great deal of choice up to implementations and combined with the need to be compatible with existing libraries, architectures and operating systems this means that a great deal of platform specific information is passed through Clang and on into LLVM bitcode. This means that, while the C Specification [13] and the accompanying POSIX portable operating system interface standard [11] ensure some level of source portability, the resulting bitcode is not generally portable in of itself.

Let us discuss a brief example, consider the ucontext_t structure, defined by POSIX as part of signal handling "to refer to the receiving thread's context that was interrupted when the signal was delivered"[11]. Listing 4 shows the definition in the Musl implementation of the standard library for i386 and powerpc. We can see from an LLVM perspective bitcode perspective that once lowered these are not remotely compatible types. Putting aside other concerns such as size, the order of fields is not the same and *getelementptr*, which is used to get a pointer into the structure, computes this offset based on the field number. Since the fields are in different locations, depending on which architectures platform header was present when we compiled the source we will get very different LLVM bitcode, even if the input program is the same.

This is partially what motivated us to present a virtual target and split standard library abstraction layer approach in our implementation. Since we are targeting current containerized applications it is useful to be compatible with C/POSIX to support unmodified legacy applications but we need to abstract away these architecture specific details if we are to be fully bytecode portable.

JIT Technology with C/C++ IR

The approach laid out in "JIT Technology with C/C++ IR"[24], discussed in section 2.2.3, attempts to address some of the optimization cases we wish to target by considering the heterogeneous nature of hardware, microarchitectural portability and how best to optimize for emerging hardware features.

[4]Clang is the C/C++/Objective-C frontend developed by the LLVM-Project.

```
// i386
typedef struct __ucontext {
        unsigned long uc_flags;
        struct __ucontext *uc_link;
        stack_t uc_stack;
        mcontext_t uc_mcontext;
        sigset_t uc_sigmask;
        unsigned long __fpregs_mem[28];
} ucontext_t;


// powerpc
typedef struct __ucontext {
        unsigned long uc_flags;
        struct __ucontext *uc_link;
        stack_t uc_stack;
        int uc_pad[7];
        mcontext_t *uc_regs;
        sigset_t uc_sigmask;
        int uc_pad2[3];
        mcontext_t uc_mcontext;
} ucontext_t;
```

Listing 4: Ucontext_t as defined by the Musl C Standard Library

Many of the difficulties we are hoping to address in the container-based cloud are similar. Our consideration for microarchitectural optimization approaches and the dynamic optimization cases we consider are inspired by their work. However as their tooling is based on IBM proprietary tools we cannot directly make use of it. We note their approach considers the case of only a single node optimizations, where as we consider both single node and cloud scale approaches. Their resulting fat binaries also correspondingly do not make a claim to be architecturally agnostic/portable, aiming rather for portability and optimization at the micro-architectural level (they discuss their rational for this and it is inline with their intended use case as outline above).

They make a key observation that also holds true for the container-based cloud:

> "The vast move to cloud and virtual environments results in increased abstraction of performance-critical information from the static compilation environment. Only upon runtime deployment do the actual physical resources become known, and these can change during program execution due to workload migration and consolidation considerations, requiring continuous online adaptation of programs."[24]

They attempt to address this with JIT techniques adapted from the JVM. In our case however we note that that with the use of orchestration systems to manage them the actual physical resources generally do become known after build time but before actual container runtime, leading to our preference for an

AOT approach which avoids some of the potential performance pitfalls and scalability costs of JIT.

Their reliance on binary images to offset the startup costs of performing JIT has obvious difficulties in an architecturally heterogeneous cloud set, and we argue is not as essential in the container-based cloud case, as there are other ways to offset this cost such as the opportunities to perform idle time AOT and caching between uploading of an image to a repository and runtime in the cloud, especially for frequently used images. Accordingly the use of a fat binary and JIT approach may not be an ideal solution in a heterogeneous cloud setting.

Portable Native Client

Portable Native Client's (pNaCl) LLVM-based compiler frontend and translator as well as virtual ABI form a central pilar of our approach. However, their implementation can be simplified by the fact that they are targeting the Native Client service runtime in a web browser rather that a normal operating system interface. While this increases their overall portability, the service runtime is intended as an isolation mechanism which is unneeded in a container context as operating system virtualization is already fulfilling that role. Where as the service run-time incurs around a 5% overhead, our split library approach occurs at link and translation time and is optimized as part of the overall application, which results in no notable overhead in comparison with standard native libraries (as we demonstrate in chapter 4).

The Man O' War system uses an almost identical virtual ABI that is derived directly from that of PNaCl. Parts of the PNaCl toolchain and source code (licensed under an Open Source license) along with their fork of LLVM served as the basis of our own compiler frontend and translator. We note there are some key difference with our overall system however.

Their portability problem is simplified somewhat, by targeting a ABI which is implemented by the browser rather than a real operating system, and thus is highly platform agnostic and controlled to their purpose as we have seen above. In order to be able to be usable for it's intended purpose our implementation must be able to run against an unmodified OS ABI. In our case this will be Linux and the Linux ABI is certainly not uniform across architectures. Our implementation discards many components intended solely for the browser and it's security model and we implemented our own novel split C library approach (described in chapter 3) to support the virtual target and overcome these difficulties on a traditional operating system. As they are focused on a client computing model, specifically that of the web, they also choose to make use of JIT techniques restricting the possible optimizations as it can be difficult to mitigate the cost of heavy optimizations on the client side (increased client-side compilation times will lead to very noticeable page load delays) where as we prefer AOT techniques for reasons discussed in the preceding section.

Java Bytecode and JVM

The JVM is a process virtual machine which is designed to execute a Bytecode format (as described in section 2.2) originally designed for the Java programming language but now targeted by a growing family of languages. Process virtual machines construct a managed runtime environment based on a virtual

instruction set tailored to the language they are intended to run, abstracting it away from the hardware platform. In order to achieve acceptable performance JIT compilation techniques are frequently adopted to compile frequently used and critical sections of the application.

In many ways this approach share some similarities with ours, bytecode-based container images resemble JAR files in some ways and Java-based PaaS exist in the cloud space. However, the types of languages and level of exposure to the hardware normally found on these systems are radically different. Applications utilising the JVM are known for having large startup costs due to the time required to JIT important paths in the code on startup [24].

In the cloud, this would involve notable costs on a per-node basis without a specialized PaaS or other techniques to offset this. This was one of the reasons we choose to favour AOT translation and aggressive caching. It is worth noting that newer JVM projects such as Graal from Oracle and the Android Runtime (ART) from Google have recently started supporting AOT compilation and caching so this may become an interesting area of future work however [16].

4.3.3 Discussion

In this final section we make some overall remarks on our approach and implementation in comparison to overall trends and reflect on design and testing choices and limitations. In contrast to some of the other frameworks we have discussed, we are not focusing on data parallel programs. We also do not focus on addressing the scheduling problem in this work.

While we attempt to be portable, we do not attempt to be operating system agnostic, in fact the opposite as our current implementation is limited to the Linux ABI using it's userspace api as a unifying force across heterogeneous platforms. We feel this harmonizes well with the container-based cloud, since os-level virtualization occurs as the syscall layer and while container implementations exist for non-Linux platforms[5], the majority of of container implementations and container-based clouds run on top of Linux. Popular container engines such as Docker originated on the Linux platform and most public container services are designed with Linux containers in mind.

We target the C language as it is the language of the Linux kernel and a great majority of application software written for the Linux platform. Since one of the major advantages of containers is being able to containerize existing Linux applications and libraries, this make for an obvious target. Even the run times for many higher level languages are implemented in C, so this is a useful target language even for those use cases. Future version of the toolchain could be extend to support other statically compiled languages in a similar fashion.

We want a program representation that is both architecture and microarchitecture agnostic as far as possible to provide the maximum opportunities for optimization in the heterogeneous cloud as seamlessly as possible. We adopt a number of components including the virtual ABI, LLVM bitcode format and translator from PNaCl but re-target it to our own split library implementation on top of an unmodified operating system. This implementation combines some of the approach and as well as heavily modified tools from the Portable Native Client system with approaches of several other bytecode systems such as

[5]Indeed some early work on containerization was done in systems such as Solaris Zones [29].

the Heterogeneous System Architecture's BRIG framework for portability of parallel applications [42].

We choose to retain the 32-bit nature of the virtual ABI (Native Client mandated/required this in their runtime). Our motivation for this choice was largely to support some of our experiments which would only run with 32-bit platforms. This comes largely from the immaturity of 64-bit arm support on the testing platforms available to us (eg. SPEC is built for ARMv7 by default) as Aarch64 (64-bit Armv8) software support is not yet standardly supported on many SOC even with Armv8 cores, though this situation is rapidly improving. Since we wish to perform migration tests from x86 to ARM, this required limiting our tests to the 32-bit architectures, though all these approaches discussed could equally be applied to 64-bit architectures. Similarly we consider only the case of little endian architectures as those are currently some of the most popular and are available to us in the cloud. Many others have some bi-endian support, often configurable by process, so this implementation can be applied there as well.

Our earlier experiments lead us to conclude that the overall impact of any overheads added by the split C library and abstraction layer appears to be very minimal. Even for a program which is I/O heavy and consequently makes many library and system calls we do not expect a significant change as our implementation does not in fact necessitate adding additional abstraction-levels to I/O functions, as these are largely independent anyway [8].

In the next chapter we will consider some directions for future work, possible extensions to the system, as well as our conclusions based on the results and discussion presented here.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

As we reach the performance limits of modern microprocessor technologies, computing resources are increasingly becoming heterogeneous [42]. In the public cloud various types of accelerators are proliferating and new hardware designs such as the ARM-based AWS Graviton processor are making the traditionally architecturally homogeneous cloud increasingly heterogeneous [1]. We have discussed the advantages of a heterogeneous cloud for being able to exploit hardware at a variety of performance vs power design points for activities such as "heterogeneous scaling" and utilizing specialized hardware extensions and accelerators [31].

There are some strong barriers to developers creating applications to make use of such resources without significant software development effort including creating highly specialized versions of their applications. Users of computing services provided by container-based cloud providers are required to provide their application in a format the infrastructure/platform can utilize, typically as a set of OCI or Docker containers containing Linux binaries. In practice, this means application developers and Independent Software Vendor (ISV) compile generic, "one size fits all" versions of their container-based applications that can generally run on any commonly available x86 infrastructure. However, this is is not necessarily the most efficient way both cost and performance wise and makes exploiting heterogeneity difficult.

We consider this a notable shortcoming as developers turn to container systems to help isolate them from the particulars of the underlying node and software configuration [35]. In order to extend containerized applications to heterogeneous systems without costly and inflexible customization by the developer an additional framework is required on top of what is normally provided by container systems. A unified approach and heterogeneous systems framework for use with container-based application virtualization in the presence of a potentially heterogeneous cloud environment is required. One which allows the application images to remain independent of any particular system but allows running applications to maximally exploit the particular features of the execution environment they are assigned to.

5.1.1 Our Contributions

We introduce a design for and partial implementation of a heterogeneous systems framework on top of containers for use in the container-based cloud to address these shortcomings and ease software development in the heterogeneous cloud. As with all heterogeneous systems frameworks portability is a major concern. We have shown that we can bridge some of the barriers present with traditionally compiled containers with a bytecode-based container system, drawing on several established techniques but tailoring them specifically to the needs of the container based cloud. This design combines some of the approaches of several other bytecode systems such as the Heterogeneous System Architecture's BRIG framework for portability of parallel applications but shifts focus to the heterogeneous cloud as a whole rather than specifically just accelerating parallel computations.

We have introduced our implementation of this design (that we refer to as Man O' War), a system based on LLVM bitcode and apply it to the standard OCI container format to produce a portable containers suitable for use with unmodified OCI compliant runtimes. Our implementation of this system creates container images containing binaries in a LLVM-based bitcode format, and then allows them to be translated on demand to the the specific target architecture and microarchitecture of the host by a tool we refer to as the container finalizer. Applications are built using a intermediate, virtual ABI (i.e le32-manowar) and a split standard library which adapts calls to this virtual ABI to the native ABI of the Linux kernel on the target platform. This system specifically allows for AOT compilation as part of a container orchestration system customarily found in a container-based cloud with aggressive caching of built images possible. Resulting container images run directly upon unmodified container systems, a novel result that eases integration and adoption with existing applications and services in the container based cloud.

We considered and performed a number of performance experiments which validate the utility of this design on systems with diverse architectural and microarchitectural configurations in the heterogeneous cloud. We demonstrated that allowing for seem-less architectural migration can lead to real cost benefits in certain heterogeneous scaling scenarios. We also demonstrated that our implementation does not have significant runtime over heads compared to a traditionally compiled C-based containerized application for a typical test workload. This allows application developers to consider it's universal application, rather than making it just another specialized tool. We also demonstrated that in is possible to dynamically incorporate information about architectural and microarchitectural heterogeneity into the compilation environment for bytecode containers during code generation, allowing the container image to make use of new hardware features.

By making these capabilities available transparently to developers of cloud-native applications and the container ecosystem we remove the burden of handling and optimizing for heterogeneity from the developer. This opens up the possibilities of exciting new dynamic optimizations and performance enhancements in the container-based cloud. We hope to see many new container-based applications, orchestrator plugins, schedulers and other works take advantage of these capabilities in the expanding heterogeneous cloud.

5.2 Future Work

Container technology research is still at an early stage [27] and while we have demonstrated some promising results for using a heterogeneous systems framework based on bytecode-based portable containers to improve resource utilization and performance optimization in the heterogeneous cloud, there are many opportunities both to expand our approach and to connect it to other works in this area. In this section we discuss possible future work and directions.

5.3 Scheduling

While we have focused primarily on the portability requirement of a heterogeneous systems framework for the container-based cloud, deferring the responsibility of generating schedules to container orchestrators and other works, generating these would be of great utility to a complete system. Utilizing information from the Man O' War toolchain and a heterogeneity-aware scheduling algorithm as set out in these works it would be interesting to investigate if we can generate deployment configurations and scheduling plugins for common orchestrators such as Kubernetes.

5.3.1 Feedback-Directed Optimization (FDO)

According to Smith "Feedback-directed optimization (FDO) is a general term used to describe any technique that alters a program based on information gathered at run time"[34]. Feedback directed compilation is an FDO method which uses runtime profiling information to guide optimization of the program during compilation [34].

The Linux kernel supports performance profiling through the it's perf performance profiling interface (which we used during our performance experiments). Tools such as Google's AutoFDO can turn these profiling traces into a binary profile which can then be utilized LLVM's built in feedback directed compilation infrastructure. The Man O' War translator can easily extended to accept this information provided this data was collected by nodes. An high level overview of how such a system might be constructed is featured in Figure 5.1 with performance data being periodical fed back into the finalizer and updated container images being deployed and cached in the image repository.

In a large cloud setting it is often the case that many containers are created from copies of the same container image, often with a similar workload. Consider for example a collection of web services running copies of popular web servers such as Nginx or Apache2. By occasionally instrumenting and profiling these containers and then provided the profiles to the Man O' War finalizer to rebuild their container images we may be able to dramatically boost performance for these types of frequently used images.

5.3.2 Extensions to Other Statically-typed Languages

One of the difficulties that motivates our split library approach is that of the leaky abstractions in the C programming language. As we have seen earlier much of this information leaks into our bytecode via the information passed by Clang, thus necessitating a split library approach. This has been a motivating

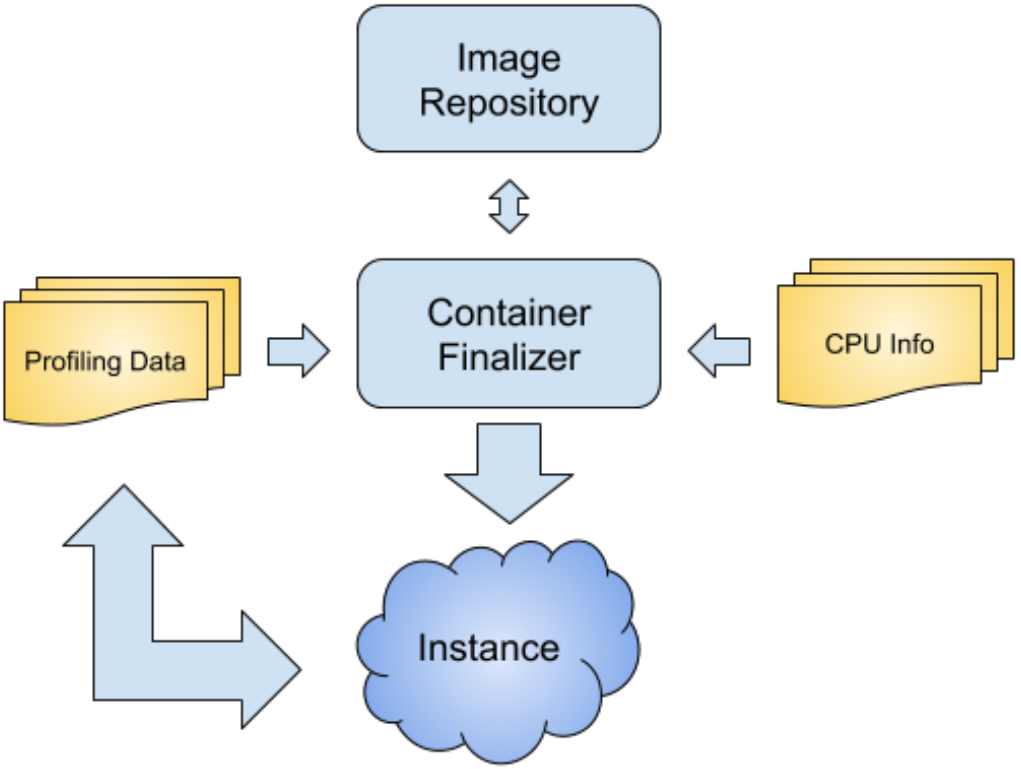


Figure 5.1: A Feedback-Directed Optimization (FDO) extension to the container system

factor in other bytecode-based system using primarily much newer language designs with stronger type systems and abstractions. Future work should include exploring how the bytecode containers technique and tools can be expanded to include other popular statically compiled languages (eg. Go and Rust) that also include new language features which may be beneficial in the presence of heterogeneity (such as, for example, Go's build-in support for concurrency and garbage collection).

5.3.3 Extensions to Other Bytecode IR

Another difficulty we found is that the LLVM type system, while quite adequate for it's current use as a compiler intermediate language, is not powerful enough to easily express all types that might be of interest to us. For example, if we wanted an approach which targeted both 32-bit and 64-bit systems it would be very helpful to have, for example, an integral type equal to the platform word size. This could possibly create more problems than it solves, as C has this sort of type with int, and this is often the source of portability problems and overflows if used improperly but it would still be of interest to see what extensions could be made to the bytecode to aid our purpose. Other extensions might include better support for certain newer concurrency constructs such as transactional memory.

5.3.4 Extension to Parallel Accelerators

Our approach and the Man O' War system currently makes it easier to exploit heterogeneity based on architecture or microarchitecture in cloud environments, but these are not the only types of heterogeneous processing resources. As discussed in chapter 2, parallel accelerators such as GPGPUs are often found in the cloud. Integrating systems such as the Heterogeneous System Architecture (HSA) which seek to provide a similarly portable environment for accelerators based on a shared-memory model [42] and unified programming model would add this dimension to our approach and is certainly an area of interest. The HSAIL IR representation and our bytecode-based executable are complementary techniques and merging them is certainly possible. This would mean extending or superseding the BRIG format as our binaries are currently not stored in an ELF container when in bytecode form, but this is certainly not an insurmountable challenge and could lead to many benefits for accessing parallel and many-core processor designs.

Bibliography

[1] [n. d.] Amazon ECS - run containerized applications in production. Amazon Web Services, Inc. Retrieved 08/08/2018 from https://aws.amazon.com/ecs/.

[2] [n. d.] Azure kubernetes service (AKS) | microsoft azure. Retrieved 08/08/2018 from https://azure.microsoft.com/en-us/services/kubernetes-service/.

[3] Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan, and Karthikeyan Sankaralingam. 2015. ISA wars: understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures. *ACM Trans. Comput. Syst.*, 33, 1, (March 2015), 3:1–3:34. ISSN: 0734-2071. DOI: 10.1145/2699682. Retrieved 07/23/2018 from http://doi.acm.org/10.1145/2699682.

[4] Theo D'Hondt. 2008. Are bytecodes an atavism? *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5146 LNCS, 140–155. DOI: 10.1007/978-3-540-89275-5-8.

[5] [n. d.] Docker. Retrieved 12/11/2018 from https://www.docker.com/index.html.

[6] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. 2010. Pnacl: portable native client executables. *Google White Paper*.

[7] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolás Serrano. 2016. DevOps. *IEEE Software*, 33, 3, (May 2016), 94–100. ISSN: 0740-7459. DOI: 10.1109/MS.2016.68.

[8] Linux Foundation. [n. d.] Linux standard base specification 5.0. Retrieved 07/28/2018 from https://refspecs.linuxfoundation.org/lsb.shtml.

[9] [n. d.] Google kubernetes engine | kubernetes engine. Google Cloud. Retrieved 08/08/2018 from https://cloud.google.com/kubernetes-engine/.

[10] Brendan Gregg. 2016. The flame graph. *Commun. ACM*, 59, 6, (May 2016), 48–57. ISSN: 0001-0782. DOI: 10.1145/2909476. Retrieved 12/07/2018 from http://doi.acm.org/10.1145/2909476.

[11] 2018. IEEE standard for information technology–portable operating system interface (POSIX(r)) base specifications, issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) - Redline*, (January 2018), 1–6900.

[12] [n. d.] ILP32 for AArch64 whitepaper. Retrieved 11/26/2018 from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0490a/ar01s01.html.

[13] 2011. ISO/IEC 9899:2011 information technology – programming languages – c. (2011). https://www.iso.org/standard/57853.html.

[14] Michael Kaufmann and Kornilios Kourtis. 2017. The HCl scheduler: going all-in on heterogeneity. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/hotcloud17/program/presentation/kaufmann.

[15] Nane Kratzke and Peter-Christian Quint. 2017. Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. *Journal of Systems and Software*, 126, (April 1, 2017), 1–16. ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.01.001. Retrieved 06/27/2018 from http://www.sciencedirect.com/science/article/pii/S0164121217300018.

[16] Paul Krill. 2017. Java 9's AOT compiler: use at your own risk. *InfoWorld.com; San Mateo*, (April 28, 2017). Retrieved 08/14/2018 from http://search.proquest.com/docview/1892960773/abstract/184A12F52F2B4BF1PQ/1.

[17] Paul Krill. 2017. WebAssembly wins! google pulls plug on PNaCl. *InfoWorld.com; San Mateo*, (June 2, 2017). Retrieved 03/19/2018 from https://search-proquest-com.ezproxy.lib.ryerson.ca/docview/1905174034/abstract/2FF3AD6DCD404488PQ/1.

[18] Chris Lattner and Vikram Adve. 2004. 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* (CGO '04). IEEE Computer Society, Washington, DC, USA, 75–. ISBN: 978-0-7695-2102-2. Retrieved 11/14/2017 from http://dl.acm.org/citation.cfm?id=977395.977673.

[19] [n. d.] Linux containers. Retrieved 12/11/2018 from https://linuxcontainers.org/.

[20] Peter Mell and Tim Grance. 2011. SP 800-145, the NIST definition of cloud computing. (September 2011). Retrieved 07/11/2018 from https://csrc.nist.gov/publications/detail/sp/800-145/final.

[21] [n. d.] Musl libc. Retrieved 07/28/2018 from https://www.musl-libc.org/.

[22] Daniel Nemirovsky, Nikola Markovic, Osman Unsal, Mateo Valero, and Adrian Cristal. 2015. Reimagining heterogeneous computing: a functional instruction-set architecture computing model. *IEEE Micro*, 35, 5, 6–14. ISSN: 0272-1732. DOI: 10.1109/MM.2015.109. Retrieved 07/18/2018 from http://journals.scholarsportal.info/details/02721732/v35i0005/6_rhcafiacm.xml.

[23] 2018. New – EC2 instances (a1) powered by arm-based AWS graviton processors. Amazon Web Services. (November 26, 2018). Retrieved 12/13/2018 from https://aws.amazon.com/blogs/aws/new-ec2-instances-a1-powered-by-arm-based-aws-graviton-processors/.

[24] Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos. 2013. JIT technology with c/c++: feedback-directed dynamic recompilation for statically compiled languages. *ACM Trans. Archit. Code Optim.*, 10, 4, (December 2013), 59:1–59:25. ISSN: 1544-3566. DOI: 10.1145/2555289.2555315. http://doi.acm.org/10.1145/2555289.2555315.

[25] 2017. Open container initiative image format specification. (July 19, 2017). https://github.com/opencontainers/image-spec/releases/tag/v1.0.0.

[26] [n. d.] Open containers initative. Open Containers Initiative. Retrieved 12/11/2018 from https://www.opencontainers.org/.

[27] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. 2018. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, 1–1. ISSN: 2168-7161. DOI: 10.1109/TCC.2017.2702586.

[28] Max Plauth, Florian Rösler, and Andreas Polze. 2018. CloudCL: distributed heterogeneous computing on cloud scale. In *Proceedings - 2017 5th International Symposium on Computing and Networking, CANDAR 2017*. Volume 2018-January. Institute of Electrical and Electronics Engineers Inc., 344–350. ISBN: 978-1-5386-2087-8. DOI: 10.1109/CANDAR.2017.49. https://www.scopus.com/inward/record.uri?eid=2-s2.0-85050285718&doi=10.1109%2fCANDAR.2017.49&partnerID=40&md5=d20132f42512d1953a03cc78caf872ea.

[29] Daniel Price, Andrew Tucker, and Sun Microsystems. 2004. Solaris zones: operating system support for consolidating commercial workloads, 14.

[30] Nathalie Rauschmayr and Achim Streit. 2013. Evaluation of x32-ABI in the context of LHC applications. *Procedia Computer Science*, 18, 2233–2240, Complete. ISSN: 18770509. DOI: 10.1016/j.procs.2013.05.394. Retrieved 11/26/2018 from http://journals.scholarsportal.info/details/18770509/v18icomplete/2233_eoxitcola.xml.

[31] Eduardo Roloff, Matthias Diener, Emmanuell D. Carreño, Francis B. Moreira, Luciano P. Gaspary, and Philippe O.A. Navaux. 2017. Exploiting price and performance tradeoffs in heterogeneous clouds. In *Companion Proceedings of the10th International Conference on Utility and Cloud Computing* (UCC '17 Companion). ACM, New York, NY, USA, 71–76. ISBN: 978-1-4503-5195-9. DOI: 10.1145/3147234.3148103. Retrieved 08/05/2018 from http://doi.acm.org/10.1145/3147234.3148103.

[32] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean Philippe Martin, and Dennis Fetterly. 2013. Dandelion: a compiler and runtime for heterogeneous systems. In *SOSP 2013 - Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 00100, 49–68. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522715. https://www.scopus.com/inward/record.uri?eid=2-s2.0-84889679621&doi=10.1145%2f2517349.2522715&partnerID=40&md5=b05e1677527e657a791f05e340db15b9.

[33] Jyoti Sahni and Deo Prakash Vidyarthi. 2017. Heterogeneity-aware adaptive auto-scaling heuristic for improved QoS and resource usage in cloud environments. *Computing*, 99, 4, (April 1, 2017), 351–381. ISSN: 1436-5057. DOI: 10.1007/s00607-016-0530-9. Retrieved 11/27/2018 from https://doi.org/10.1007/s00607-016-0530-9.

[34] Michael D. Smith. 2000. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization* (DYNAMO '00). ACM, New York, NY, USA, 1–11. ISBN: 978-1-58113-241-0. DOI: 10.1145/351397.351408. Retrieved 08/03/2018 from http://doi.acm.org/10.1145/351397.351408.

[35] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (EuroSys '07). ACM, New York, NY, USA, 275–287. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273025. Retrieved 08/08/2018 from http://doi.acm.org/10.1145/1272996.1273025.

[36] [n. d.] SPEC CPU2017 documentation. Retrieved 08/08/2018 from https://www.spec.org/cpu2017/Docs/.

[37] 2014. SPIR - the industry open standard intermediate language for parallel compute and graphics. The Khronos Group. (January 20, 2014). Retrieved 12/05/2018 from https://www.khronos.org/spir/.

[38] John E. Stone, David W. Gohara, and Guochun Shi. 2010. OpenCL: a parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12, 3, (May 2010), 66–73. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.69.

[39] [n. d.] Taints and tolerations - kubernetes. Retrieved 07/19/2018 from https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/.

[40] 2018. The path to GPU as a service in kubernetes. In collaboration with Viraj Chavan and Renaud Gaubert. NVIDIA GTC 2018 - San Jose, (2018). Retrieved 12/03/2018 from http://on-demand.gputechconf.com/gtc/2018/video/S8893/.

[41] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. 2012. Alsched: algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing* (SoCC '12). ACM, New York, NY, USA, 25:1–25:7. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391254. Retrieved 08/05/2018 from http://doi.acm.org/10.1145/2391229.2391254.

[42] Wen-mei W. Hwu, editor. Heterogeneous System Architecture: A New Compute Platform Infrastructure. (1 edition edition). Morgan Kaufmann, (November 20, 2015).

[43] Stuart A. West and E. Toby Kiers. 2009. Evolution: what is an organism? *Current Biology*, 19, 23, (December 15, 2009), R1080–R1082. ISSN: 0960-9822. DOI: 10.1016/j.cub.2009.10.048. Retrieved 07/26/2018 from http://www.sciencedirect.com/science/article/pii/S0960982209019101.

[44] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53, 1, (January 2010), 91–99. ISSN: 0001-0782. DOI: 10.1145/1629175.1629203. Retrieved 03/19/2018 from http://doi.acm.org/10.1145/1629175.1629203.

[45] Bennet Yee, David C. Sehr, G. Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: a sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*. 2009 30th IEEE Symposium on Security and Privacy. (May 2009), 79–93. DOI: 10.1109/SP.2009.25.

Glossary

cgroups A Linux kernel process resource limiting and isolation mechanism. A contraction of of control groups. 17

Fat Binary An executable which contains code for multiple target architectures. 12, 17

Intermediate Representation (IR) An intermediate program form in a compiler system. Often a bytecode. 17, 69

Just-in-Time Compilation (JIT) A technique consisting of on-demand compile sections of a program as they are executed, often used in the context of a VM or an interpreted language for performance reasons. 12, 69

Utility computing A pay-as-you-go service model for computing that can be purchased as if from a traditional utility such as power or water. Often attributed to John McCarthy.. 1

Acronyms

ABI Application Binary Interface. 24, 25, 27

AOT Ahead-of-Time. 12, 51, 52, 56

AVX2 Advanced Vector Extensions 2. 41

CPU Central Processing Unit. 12

CSP Cloud Service Provider. 23

DSP Digital Signal Processor. 12

GPU Graphics Processing Unit. 6, 12, 20

HPC High Performance Computing. 4, 16

HSA Heterogeneous System Architecture. 23

IaaS Infrastructure as a Service. 17, 18, 22

IR Intermediate Representation. 17, 23, 48, *Glossary:* Intermediate Representation (IR)

ISA Instruction Set Architecture. 2

ISV Independent Software Vendor. 55

JIT Just-in-Time. 12, 14, 17, 50–52, *Glossary:* Just-in-Time Compilation (JIT)

JVM Java Virtual Machine. 13, 50, 51

LSB Linux Standard Base. 25

LTO Link Time Optimization. 14

NUMA Non-Uniform Memory Access. 20

OaaS Optimization-as-a-Service. 23, 31, 32

OCI Open Container Initiative. 5, 23, 28, 56

PaaS Platform as a Service. 17, 18, 22, 23, 31, 32, 52

POSIX Portable Operating System Interface. 24

RAM Random Access Memory. 36

SIMD Single Instruction Multiple Data. 6

SLA Service Level Agreement. 11

SOC System-on-Chip. 38, 53

VM Virtual Machine. 15, 67