BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

CUSTOMIZATION AND REUSE IN DATACENTER OPERATING SYSTEMS

by

DAN SCHATZBERG

B.A., Boston University, 2010

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2017

Approved by

First Reader ____
Jonathan Appavoo, Ph.D.
Associate Professor of Computer Science

Second Reader ____
Richard West, Ph.D.
Professor of Computer Science
Professor of Electrical and Computer Engineering

Third Reader ____
Hongwei Xi, Ph.D.
Associate Professor of Computer Science

CUSTOMIZATION AND REUSE IN

DATACENTER OPERATING SYSTEMS

DAN SCHATZBERG

Boston University, Graduate School of Arts and Sciences, 2017

Major Professor: Jonathan Appavoo, Ph.D., Associate Professor of Computer Science

ABSTRACT

Increasingly, computing has moved to large-scale datacenters where application performance is critical. Stagnating CPU clock speeds coupled with increasingly higher bandwidth and lower latency networking and storage puts an increased focus on the operating system to enable high-performance.

The challenge of providing high-performance is made more difficult due to the diversity of datacenter workloads such as search, video processing, distributed storage, and machine learning tasks. Our existing general purpose operating systems must sacrifice the performance of any one application in order to support a broad set of applications.

We observe that a common model for application deployment is to dedicate a physical or virtual machine to a single application. In this context, our operating systems can be specialized to the purposes of the application.

In this dissertation, we explore the design of the Elastic Building Block Runtime (EbbRT), a framework for constructing high-performance, customizable operating systems while keeping developer effort low. EbbRT adopts a lightweight execution environment which enables applications to directly manage hardware resources and specialize their system behavior. An EbbRT operating system is composed of objects called Elastic Building Blocks (Ebbs) which encapsulate functionality so it can be incrementally extended or optimized. Finally, EbbRT adopts a unique heterogeneous and distributed architecture where an application can be split between a server running an existing general purpose operating system and a server running a customized library operating system. The library operating system provides the mechanisms for application execution including primitives for event driven programming, componentization, memory management and I/O.

We demonstrate that EbbRT enables memcached, an in-memory caching server, to achieve more than double the performance with EbbRT than with Linux. We also demonstrate that EbbRT can support more full-featured applications such as a port of Google's V8 javascript engine and nodejs, a javascript server runtime.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Operating systems have two primary functions: 1. Securely multiplex hardware resources across multiple applications and users and 2. Provide common abstractions and routines which reduce the burden of application development. Operating systems abstractions typically serve both purposes. Consider *threads* which, on the one hand enable the operating system to multiplex a processor across multiple applications, and on the other hides the hardware parallelism from the application developer. Similarly, virtual memory and sockets both enable isolation and hide hardware details.

The drawback to these abstractions is that they prevent optimizations which enable high performance. This has been exacerbated by several recent trends:

1. CPU clock speeds have stalled in comparison to gains in networking speeds (10 Gigabit ethernet (GBE) is ubiquitous in datacenters with 50 and 100 GBE on the horizon) and high-performance storage technologies such as flash and non-volatile RAM. This puts pressure on the software I/O stack to perform well.

2. Increasingly, computation occurs in large datecenters where small, per-machine performance wins result in substantial savings in aggregate. Developers are willing to spend significant engineering effort in order to improve the performance of large-scale applications.

3. Our existing abstractions are a poor fit for the diverse set of hardware that exists: threads abstract too much of the necessary details in an enviornment with NUMA and heterogeneous multiprocessors, sockets are a poor fit for RDMA technologies,

and new programming paradigms are created to manage accelerators such as GPUs and FPGAs.

Fundamentally, high performance software is enabled by the ability to customize application workloads to the hardware characteristics. Abstractions inherently restrict the ability for application developers to customize to hardware characteristics. The desire for high-performance in datacenter software has led to the development of techniques like 1. library operating systems, where system functionality is linked into the application address space, 2. hardware virtualization, where hardware enforced isolation enables operating systems to expose hardware interfaces directly to the application, or 3. kernel bypass, where applications can interact directly with hardware. These techniques enable applications developers to customize system functionality and thereby optimize their applications to the characteristics of the hardware. However, these techniques have not seen widespread use in application development. An inherent problem is that these techniques put an increased burden on the application developer to replicate functionality provided by higher-level abstractions. Furthermore, these techniques do not provide any path for incremental development. Applications must be largely rewritten to target low-level interfaces.

Throughout the history of operating systems, this tension between high-level abstractions which provide rich functionality and and allow for a wide-range of applications to be developed, and low-level abstractions which enable specialization has existed. Typically, operating systems support some fixed interface which makes a tradeoff between the performance achievable by any one application and the support provided to developers through high-level abstractions and common functionality. For example, general purpose operating systems like Linux support a wide-range of applications with relatively high-level abstractions whereas specialized operating systems like IBM's CNK [69] target a restricted set of applications (high-performance computing) and are able to achieve significantly higher performance.

We do not believe that an operating system must make a static tradeoff between performance and developer support. Rather, we believe that, with an appropriate design, system software can be made which is high-performance yet also provides support for a wide-range of applications. We explore the design of operating systems for datacenter applications which enable high performance through customization yet encourage the reuse of existing software and software abstractions in order to reduce developer effort. Specifically, we present the design and implementation of the Elastic Building Block Runtime (EbbRT), a *framework* for constructing per-application library operating systems. EbbRT reduces the effort required to construct and maintain library operating systems without restricting the degree of specialization required for high performance. We present three novel contributions which enable EbbRT to achieve its goals.

1. EbbRT is comprised of a set of components, called Elastic Building Blocks (Ebbs), that developers can extend, replace, or discard in order to construct and deploy a particular application. This enables a greater degree of customization than a general purpose system and promotes the construction of reusable software.

2. EbbRT uses a lightweight, event-driven execution environment that allows application logic to directly interact with hardware resources, such as memory and I/O devices.

3. EbbRT applications are distributed across both specialized library operating systems and general purpose operating systems. This allows functionality to be offloaded, which reduces the engineering effort required to port applications.

In the remainder of this chapter we provide additional background and present an overview of the EbbRT design. In Chapter 2 we discuss related work in the research community. Chapter 3 presents our low-level execution environment. Chapter 4 presents our techniques for enabling modularity within EbbRT and in particular describes the Elastic Build Block. Chapter 5 discusses the manner in which EbbRT provides compatibility with existing software interfaces and specifically describes our distributed architecture. Chapter 6 presents our experimental evaluation of EbbRT and finally, Chapter 7 concludes and discusses potential future work.

1.1 Background

In this section, we provide broader context for this work with a particular focus on observations of datacenter hardware trends as well as patterns in application development. These observations motivate the need for both high-performance and reduced developer effort in the construction of datacenter applications.

1.1.1 Computing in the Datacenter

A subtantial portion of the world's computation is performed in large datacenters containing thousands of servers with high-bandwidth (10s of gigabits) and low-latency (sub-millisecond) interconnectivity. Datacenter applications are built to span hundreds of machines. Application developers must consider new problems at scale such as fault tolerance, load balancing, sharding, and tail-latency.

The focus of software development is inherently a cost-benefit tradeoff and investments in performance improvement provide more return at scale. A developers time can be spent building new software, adding additional features, enhancing software stability or improving performance. The developer cost to make a five percent performance improvement is largely independent of the scale at which the application is deployed, whereas the benefit of performance improvements is obviously dependent on scale. Therefore, we must recognize that developers are willing to spend significant effort optimizing datacenter applications.

Datacenter applications tend to have high fan-out degree in their communication patterns. In order to process a single web request, a webserver may need to request objects from a distributed caching service. It is not rare for a single web request to require interacting with hundreds of other servers. The user-perceived latency is therefore heavily dependent on the tail of the response latency distribution of the backend services. Most of the time is spent waiting for the slowest of the backend services. One of the most significant causes of tail latency is queueing effects. A heavily loaded service may delay responses simply due to a burst of incoming requests. Queuing is more likely as the request rate approaches the service rate. In order to cope with this, datacenter services tend to under-utilize individual servers, forcing internet-scale companies to spent billions of dollars on under-utilized hardware. This is exacerbated by the fact that the power draw of modern servers is not proportional to their utilization. At 30% utilization, a state of the art datacenter will still draw 70% of it's peak power [47]. Therefore, developers are further incentivized to better utilize datacenter hardware through high-performance software.

In addition to the added value of improving application performance, there are further opportunities for optimization in a datacenter. Network communication within a datacenter is low latency and high-bandwidth and continuously improving whereas CPU clock speeds have stagnated. Similarly, disk access latencies and bandwidths have improved in datacenters. In contrast to early networked systems where I/O costs dominated, now developers are incentivized to optimize the software which interacts with an I/O device.

Additionally, datacenters tend to be fairly homogeneous platforms. A datacenter will typically only have a few generations of Intel x86 servers which get replaced within 3–5 years. Compare this to desktop or mobile environments where end-users may have hardware from many different vendors with drastically different capabilities and are much slower to upgrade. This enables and incentivizes application developers to optimize applications to the particular details of the hardware they run on.

In summary, properties of datacenter computation such as scale, platform homogeneity and tail latency create an increased focus on software performance.

1.1.2 Modern application development

One of the most substantial outcomes of cloud computing is that it has commoditized the datacenter. Developers and small businesses can get access to and build software for datacenter hardware just the same as internet-scale companies. This results in a proliferation of software, making datacenter software much more diverse than other domains such as high performance computing. Datacenter software includes workloads such as search, ads targeting, distributed storage, user-facing webservers, video processing, and machine learning tasks. These have drastically different performance characteristics and objectives. For example, user-facing webservers and the backend services they depend on are incredibly sensitive to tail-latency, whereas offline video processing or machine learning is better served by higher throughput. Webservers tend to have ample available parallelism due to their request-driven nature, whereas video processing workloads may require substantial engineering effort to enable sufficient parallelism. This leads us to believe that per-application optimization is critical to achieving high performance in datacenters.

This is further exacerbated by the fact that application software has become significantly less reliant on any single software API. For example, it has become quite rare for applications to directly write to a POSIX interface. In fact, many applications are written in managed programming languages such as Java, Go, Python, and PHP which present drastically different APIs than the lower-level system interfaces which they depend on. Optimizing the performance of any one managed programming language provides less value than optimizing a more common primitive. However, the fact that applications are increasingly being written to high-level interfaces provides a greater opportunity for optimization without the burden of compatibility with lower-level interfaces.

Another relatively recent shift in application development is the rate at which software is developed and deployed. Large internet-scale companies will routinely deploy code to user-facing services within hours or days of it being written. This rate of development and deployment makes performance optimization particularly challenging as such optimization must occur concurrently with feature development and changes in workload patterns. Expending developer effort to optimize an application that may look and behave very differently by the time the optimization is complete is wasteful. Instead, performance optimization must be a continuous and iterative process. Rewrites of software are rarely undertaken because the effort invested is so large and the outcome is uncertain.

It is important to also note that not all dependencies of an application are performance critical. Modern applications depend on libraries for auxiliary functionality such as logging or recording data for ad-hoc analysis. This functionality isn't critical to the overall performance of the application. In the process of optimizing an application, one can provide suboptimal but otherwise functional behavior for substantial portions of an application. In doing so, it may be possible to reduce the effort normally required to optimize an application.

1.1.3 Summary

These observations lead us to a number of objectives EbbRT should fulfill.

1. The properties of a datacenter both incentivize extreme optimizations. EbbRT should provide as much ability to customize system software as possible.

2. The existence of such a diverse number of platforms and software APIs makes it unrealistic to expect applications to be rewritten to new interfaces — therefore EbbRT must provide compatibility with existing interfaces, at least in part.

3. Finally, application development cannot be put on hold while performance is optimized. EbbRT must encourage incremental optimization of applications.

1.2 Design Overview

This section describes the high-level design of EbbRT. In particular the three elements of the design discussed are: 1. a heterogeneous distributed structure, 2. a modular system structure, and 3. a non-preemptive event-driven execution environment.

1.2.1 Heterogeneous Distributed Structure

Common datacenter deployment models enable a single application to be deployed across multiple machines within an isolated network. In this context, it is not necessary to run general purpose operating systems on all the machines. Rather, an application can be deployed across a heterogeneous mix of specialized library OSs and general purpose operating systems.

To facilitate this deployment model, EbbRT is implemented as both a lightweight bootable runtime and a user-level library that can be linked into a process of a general purpose OS. We refer to the bootable library OS as the *native* runtime and the user-level library as the *hosted* runtime.

The native runtime allows application software to be written directly to hardware interfaces uninhibited by legacy interfaces and protection mechanisms of a general purpose operating system. The native runtime sets up a single address space, basic system functionality (e.g. timers, networking, memory allocation) and invokes an application entry point, all while running at the highest privilege level. The EbbRT design depends on application isolation at the network layer, either through switch programming or virtualization, making it amenable to both virtualized and bare-metal environments.

The hosted user-space library allows EbbRT applications to integrate with legacy software. This frees the native library OSs from the burden of providing compatibility with legacy interfaces. Rather, functionality can be offloaded via communication with the hosted environment.

A common deployment of a EbbRT application consists of a hosted process and one or more native runtime instances communicating across a local network. A user is able to interact with the EbbRT application through the hosted runtime, as they would any other process of the general purpose OS, while the native runtime supports the performance-critical portion of the application.

1.2.2 Modular System Structure

To provide a high degree of customization, EbbRT enables application developers to modify or extend all levels of the software stack. To support this, EbbRT applications are almost entirely comprised of objects we call *Elastic Building Blocks* (Ebbs). As with objects in many programming languages, Ebbs encapsulate implementation details behind a well-defined interface.

Ebbs are distributed, multi-core fragmented objects [27, 64, 75], where the namespace of Ebbs is shared across both the native and hosted runtimes. An EbbRT application typically consists of multiple Ebb instances. The framework is composed of base Ebb types that a developer can use to construct an EbbRT application.

When an Ebb is invoked, a local *representative* handles the call. Representatives may communicate with each other to satisfy the invocation. For example, an object providing file access might have representatives on a native instance simply function-ship requests to a hosted representative which translates these requests into requests on the local file system. By encapsulating the distributed nature of the object, optimizations such as RDMA, caching, using local storage, etc. would all be hidden from clients of the filesystem Ebb.

Ebb reuse is critical to easing development effort. Exploiting modularity promotes reuse and evolution of the EbbRT framework. Developers can build upon the Ebb structure to provide additional libraries of components that target specific application use cases.

1.2.3 Execution Model

Execution in EbbRT is non-preemptive and event-driven. In the native runtime there is one event loop per core which dispatches both external (e.g. timer completions, device interrupts) and software generated events to registered handlers. This model is in contrast to a more standard threaded environment where preemptable threads are multiplexed across one or more cores. Our non-preemptive event-driven execution model provides a low overhead abstraction over the hardware. This allows our implementation to directly map application software to device interrupts, avoiding the typical costs of scheduling decisions or protection domain switches.

EbbRT provides an analogous environment within the hosted library by providing an event loop using underlying OS functionality such as `poll` or `select`. While the hosted environment cannot achieve the same efficiency as our native runtime, we provide a compatible environment to allow software libraries to be reused across both runtimes.

Many cloud applications are driven by external requests such as network traffic so the event-driven programming environment provides a natural way to structure the application. Indeed, many cloud applications use a user-level library (e.g. libevent[73], libuv[7], Boost ASIO[2]) to provide such an environment.

Further, we provide interfaces which allows application software to directly manage memory and I/O devices with little to no abstraction. These low-level interfaces are critical for enabling performance specialization.

Chapter 2

Related work

This chapter provides the larger context for this work and compares it with past systems research. In particular, we focus on the historical tension between the desire for operating systems to 1. Provide protection or isolation between applications and users, 2. Support a broad set of scenarios and applications, and 3. Enable high performance. This is presented in a mostly chronological order as most research builds on prior work.

An Overview of the CAL Time-Sharing System [54] This work, published in 1969, describes the construction of CAL-TSS, a time-sharing system for the dual-processor Control Data 6400. The primary objectives of this system are the construction of a flexible (in the sense that it can be used to support a broad range of applications) and reliable system. The author identifies one of the key principles of operating system architecture, the notion of *layering*. A system can provide "low-level" primitives on top of which, "higher-level" facilities can be constructed. It is simpler to guarantee properties (in this case reliability) of a small set of primitives rather than a large, monolithic system.

One of the key observations made in this work is that low-level primitives need not be convenient to use, each operation can do very little, but the collection must be powerful enough to allow for the construction of more convenient abstractions. Furthermore the author observes that

> If a system is to evolve to meet changing requirements, and if it is to be flexible enough to permit modularization without serious losses of efficiency, it must have a basic structure which allows extensions not only from a basic system but

> also from some complex configuration which has been reached by several prior stages of evolution. In other words, the extension process must not exhaust the facilities required for further extensions. The system must be completely open-ended, so that additional machinery can be attached at any point.

This particular lesson, that systems extensions should be extensible themselves, is one we aggressively adopt in the design of EbbRT. Rather than providing extensibility at some fixed interface, we make the extensibility of EbbRT a first class aspect.

An Open Operating System for a Single-user machine [55] This work, published in 1979, describes the construction of the operating system for the Xerox Alto. Most contemporaneous systems targeted a multi-user environment where the interface between the operating system and the application was strictly defined in order to isolate applications from each other. This system differs significantly by targeting a single-user machine and recognizing that there need not be a distinction between system functionality and application functionality.

> Thus the system can reasonably be viewed as a collection of procedures which implement various potentially useful abstract objects. There is no significant difference between these system procedures and a set of procedures which the user might write to implement his own abstract objects. In fact, the system code is made available as a set of independent subroutine packages, each implementing one of the objects,

An important observation made in this work is that the success of the operating system as a collection of procedures depends heavily on the design of the procedures themselves. In particular, the authors recognize that it is a challenge to construct components which can be made to create a cohesive system and yet are individually useful themselves.

Protocol Service Decomposition for High-Performance Networking [63] This work, published in 1993, describes the design of a networking subsystem for the Mach 3.0

microkernel. The typical way of implementing networking services in a microkernel involves a userspace server which controls all network access and provides an interface with which applications can send and receive network traffic. The authors of this work identify that such an approach provides flexibility (different network servers can be implemented on top of the same microkernel) at the expense of performance (data usually must be copied twice from an application to the userspace server and then from the server to the kernel).

This work identifies that one can achieve both flexibility and performance by implementing network protocols as a library linked into an application. Protection is enforced by the kernel and connection setup and exceptional conditions are handled by the networking server. However, the performance-critical data path is implemented almost entirely in the application address space except for low-level packet send and receive (implemented at the kernel level). Furthermore, this technique allows applications to avoid the standard socket interface. The authors demonstrate the performance advantages gained by avoiding a redundant data-copy forced by the socket interface.

This works illustrates an important aspect of EbbRT's design — by implementing system functionality as a software library, we can support existing software interfaces efficiently, yet also provide the flexibility needed to customize for the application's needs.

Extensible Multi-user Operating Systems Spin [25] (1995) explores a way to safely extend a kernel with per-application behaviors. The observation made by the SPIN authors is that a primary cause for poor performance in microkernels is that system functionality is provided by servers which reside in a different address space from the kernel and each other. The cost of an address space switch prevents fine-grain composition in a microkernel. Instead, the authors allow for extensions to be linked directly into the kernel address space.

Some modern systems take a similar approach. Linux modules can be dynamically inserted into the kernel address space. However, a misbehaving kernel module can bring down the entire system. Therefore kernel modules must only be loaded by trusted users, limiting the ability to customize OS functionality.

In SPIN, to prevent extensions from damaging the system, the kernel makes sure all extensions are written in a type-safe, garbage collected language, Modula-3. All extensions, therefore, are known statically not to access memory or interfaces they are not allowed to.

VINO (1996) takes a very similar approach to SPIN. Kernel extensions in VINO, called *grafts*, are directly linked into the kernel address space as in SPIN. However, grafts can be written in any language. VINO prevents grafts from accessing invalid memory or executing unauthorized instructions by static analysis of the binary and inserting dynamic checks into the graft machine code. The authors note that the use of the typesafe language in SPIN would have simplified many challenges they ran into. However, they note that

> the areas we found most challenging, such as detecting and dealing with resource hoarding, identifying malicious extensions and identifying the set of graft-callable and graft-replaceable interfaces are also challenges for SPIN.

A primary concern of the paper is preventing other potentially damaging actions a misbehaving graft can perform such as resource hoarding (acquiring a kernel lock and spinning) and denial of service. To this end, grafts execution within the context of a *transaction*. A transaction logs the actions taken by a graft and if a graft needs to be aborted (e.g, it takes too long), then the log can be used to back out the actions of the graft. For example, if a graft acquires a lock and spins, the kernel can abort the graft for taking too long and release the lock. The authors measure the overhead of the transaction mechanism and show that it can be quite costly indeed, almost tripling the cost of an unprotected graft in the worst case.

Both SPIN and VINO focus on mechanisms for enabling applications to safely extend the kernel with new system functionality. In EbbRT, we side step this challenge by adopting a library operating system approach where the operating system functionality is linked into the application address space and therefore is free to be modified. Furthermore, both SPIN and VINO identify the substantial challenge of identifying where to extend the operating system. As previously discussed, our approach to modularity — in particular, the fact that the entire system is composed of Ebbs, enables developers to freely extend the system.

Exokernel: An Operating System Architecture for Application-Level Resource Management [33] Exokernel, published in 1995, takes a slightly different approach than the previous systems. Exokernel recognizes the need for customization:

> Fixed high-level abstractions *hurt application performance* because there is no single way to abstract physical resources or to implement an abstraction that is best for all applications. In implementing an abstraction, an operating system is forced to make trade-offs between support for sparse or dense address spaces, read-intensive or write-intensive workloads, etc.

In contrast to microkernels which attempt to provide minimal abstractions over hardware resources (e.g, address spaces are an abstraction over page tables or a software-managed TLB), an exokernel securely exposes hardware as directly as possible to applications. Operating system functionality is then linked directly into the application's address space using a *Library Operating System.*

To accomplish this, the authors argue that an exokernel should expose allocation, allowing a library operating system to request specific resources instead of implicitly allocating them. In addition, the exokernel exposes physical names, avoiding a level of indirection and allowing library operating systems to exploit the semantics of a physical name. Finally, an exokernel makes resource revocation visible, allowing a well-behaved library operating system to choose which resources to give up.

While the interface the exokernel provides may be less efficient than others (e.g, revoking a physical page directly without consulting the application is faster), the ability to customize outweighs the costs for those applications that demand it. This is an interesting tension, the value of customizability is very dependent on the applications the system wishes to support. If one wishes to support only read-intensive workloads, then it is more efficient to optimize for that case rather than provide customizable interfaces. In turn, the need for customizability from applications is driven by the hardware. As hardware bottlenecks are relieved, software will become more significant and applications will have a greater need for customizability.

EbbRT adopts the library operating system approach described in this work but we need not rely on an explicit exokernel to multiplex the hardware resources. Instead, isolation is provided by virtualization or network switch programming which allows us to dedicate a complete physical or virtual machine to our library operating system. Our independence from a separate exokernel allows us to more directly expose hardware interfaces to application developers.

Componentized Operating Systems There have been several efforts to explore componentization of operating system functionality such as Choices [28], TinyOS [57] and OSKit [37]. Each of these focuses on the construction of operating systems from a collection of components. In particular OSKit is largely motivated by the observation that a lot of the effort in constructing a new operating system involves replicating a lot of functionality from existing operating systems.

One of the observations made by the authors of OSKit is the importance of defining interfaces. Software components in OSKit strictly define both their interface “above”, the interface to the service exported by the component as well as their interface “below”, the dependencies of the component. This allows components to be used in isolation of the rest of the system. Additionally, OSKit provide a software “glue” layer which wraps device drivers in existing operating systems such as Linux in such a way that they can be more easily incorporated into new operating systems.

OSKit also makes a number of design decisions which are incompatible with EbbRT’s objectives. The method they use to define software components, in particular their use of dynamic dispatch and a heavy-weight interface definition language (COM), sacrifices performance in exchange for flexibility. In EbbRT we seek to apply many of the same lessons learned about the composition of an OS framework, but we wish to do so without sacrificing performance.

Cloud Library Operating Systems There has been a recent resurgence in library operating systems research. In particular, a number of research groups have made the same observation we have, that isolation provided by virtualization enables one to deploy library operating systems. MirageOS [62] executes in a single address space with the application and operating system compiled into a static system image capable of executing as a Xen virtual machine.

MirageOS focuses on improving system security by using a memory and type-safe programming language, OCaml, to define the entire system. At construction time, the application and system are compiled into a single immutable system image which prevents various code injection attacks by construction.

OS$^{\text{v}}$, published in 2014, similar to MirageOS, constructs a single-address space library operating system for cloud computing applications. Rather than targeting a high-level language, OS$^{\text{v}}$ can run many Linux binaries unmodified. The system contains a dynamic loader, capable of resolving symbols at boot-time in-order to construct the single-address space.

OS$^{\text{v}}$ focuses on the performance advantages gained by constructing a system explicitly for a single-address space. For example, context switches between threads need not invalidate the TLB (and often can avoid saving volatile registers). Furthermore, they allow the application to influence system behavior in cases of memory pressure.

Both MirageOS and OS$^{\text{v}}$ take advantage of the ability to construct single address-space applications given the common deployment model of cloud applications. They differ substantially in the degree of compatibility they provide with existing applications as well as their goals (security and performance, respectively). We adopt a similar approach to these systems in constructing single address-space library operating systems but take a much more aggressive approach to per-application specialization.

2.0.1 Summary

The operating system is responsible for allowing applications to efficiently use the hardware through a common set of reusable abstractions. No single operating system structure is best suited for all scenarios. There are many different approaches to building a system with different trade-offs to be made. Throughout the history of operating systems, the common themes of protection, extensibility, and performance are at odds with each other. The "correct" trade-offs are dependent on the goals of the operating system. A system targeted to a particular workload can achieve better performance than one designed to be extended to many different applications. Past work has shown that there are many different ways to make trade-offs within a system.

In our work, we recognize that the concern of protection is largely eliminated in the domain of datacenter computing where protection is often provided by the hardware directly or through virtualization. However, we do not believe that a system must make a static trade-off between performance and extensibility. Instead we believe that, with an appropriate design, system software can be made which is high-performance and yet also extensible to a wide-range of applications.

Chapter 3

Execution Environment

An execution environment defines an applications interface to hardware resources. Often these interfaces are used to provide a level of indirection which the operating system can use to multiplex hardware resources more freely. For example, the predominant primitive for computation is a thread of execution which can be preempted to multiplex a CPU. Programming languages and their implementations also play a role in defining an application's execution environment. For example, Message Passing Interface (MPI) is a programming language standard where independent computation nodes communicate through explicit message passing. This is then mapped by the implemented onto a shared-memory multicore or a distributed system without application modification.

Advances in hardware virtualization and the resulting development of Infrastructure as a Service clouds has created a situation where the multiplexing provided by our modern operating systems is no longer a requirement. Therefore, the execution environment of a new cloud operating system need not provide multiplexing itself. Rather, it can focus on providing a thin abstraction layer that enables applications to implement their behavior at lower levels than most systems allow.

EbbRT is designed and implemented to be an extensible framework and so, its base mechanisms can be used to create a range of different execution environments. The goal is to enable execution environments that are very different to the base environment explicitly explored in this dissertation. Our approach is similar in spirit to how one might use the Unix process model to construct a very different execution environment such as the Java Virtual Machine or the Go language runtime. Despite the fact that EbbRT is open source and freely modifiable, for the purposes of this discussion, we treat the base execution environment as fixed. This defines an upper bound on the degree of customization possible within EbbRT. It is important for the base execution environment to not only allow for high-performance and optimization, but also to provide a broadly applicable environment for many applications in order to encourage the construction of common software libraries to be used within the environment.

EbbRT provides two distinct base execution environments (simply referred to as the EbbRT execution environments from here on). The *native* environment allows application code to run as privileged software with minimal abstractions over hardware interfaces. The *hosted* environment can be embedded within a process of another operating system. The hosted environment allows applications to consume functionality provided by existing operating systems. The two environments have distinct uses, but provide similar interfaces in order to promote code reuse and interoperability.

In this chapter we describe the design and implementation of both environments. In particular we focus on describing three key aspects of the execution environment: compute, memory, and I/O. For each of these aspects, we describe their design and implementation and then discuss the advantages and limitations.

3.1 Native Environment

The goal of the native execution environment is to provide the basic functionality which one can use to construct applications. In particular, the execution environment provides the ability to construct and invoke Ebbs as well as a *core* set of Ebbs designed to facilitate direct application access to hardware resources. This provides the basis on top of which all further Ebbs, applications and libraries are constructed.

The core set of Ebbs may be replaced or modified by users, but they all must exist. For example, the `GeneralPurposeAllocator` is used during initialization. The system is not dependent on its particular implementation, just that it exists and conforms to its defined interface.

The native environment of EbbRT is distributed as a modified GNU toolchain (`gcc`, `binutils`, `libstdc++`) and C library (`newlib`) which provide an x86_64-ebbrt target. Developers compile application and library code with this toolchain to produce a bootable ELF linked with the native runtime. This ELF can be booted by any multiboot compatible bootloader (e.g. `GRUB`, `QEMU`, `kexec`).

In this section we describe the EbbRT native execution environment in three main pieces 1. our abstraction for computation, events 2. memory management 3. and I/O. For each of these pieces we describe the application programmer API, the default implementation and the advantages and disadvantages of our approach.

3.1.1 Events

EbbRT has two primary requirements of our execution model. First, the primitives should provide minimal abstraction over hardware resources in order to support the broadest set of optimizations. Second, the ability to take advantage of such optimizations should not preclude the reuse of existing software. In this section, we describe *events*, our method for acheiving these objectives.

EbbRT's execution model is event-driven. Programmers define and specify *event handler* functions which execute in response to a particular triggering *event*. In an event-driven execution model, programmers do not typically write long-lived loops, rather the environment provides event dispatch loops which monitor for the occurence of events and dispatch event handlers accordingly.

When the native environment boots, an event loop per core is initialized. Events can be triggered from timer completions, device interrupts, or through an explicit software interface. While a processor is executing an event, all interrupts are disable. When an event completes, interrupts are enabled, causing more events to be processed. Such a computation model where units of computation run to completion is described as *non-preemptive*. Our event-driven execution environment can be efficiently implemented directly on top of interrupts, providing a low overhead abstraction over the hardware facilities. This allows application software to run immediately off a device interrupt without the typical costs of scheduling decisions. In chapter 6 we demonstrate that this can result in 40% latency improvements in networking microbenchmarks.

Devices can allocate a hardware interrupt from the `EventManager` and then bind a handler to that interrupt. When an event completes and the next hardware interrupt fires, a corresponding exception handler is invoked. Each exception handler execution begins on the top frame of a per-core stack. The exception handler checks for an event handler bound to the corresponding interrupt and then invokes it. When the event handler returns, interrupts are enabled and more events can be processed.

Applications can also `Spawn` synthetic events on any core in the system. The `Spawn` method of the `EventManager` receives an event handler as a parameter which is invoked from the event loop. Spawned events are only executed once. If an application wishes to have a reoccurring event handler invoked, then it may be installed as an `IdleHandler`. In order to prevent interrupt starvation, when an event completes the `EventManager`:

1. Enables then disables interrupts, providing a short window to handle any pending interrupts
2. Dispatches a single synthetic event (from `Spawn`), if one exists
3. Invokes all `IdleHandlers`
4. Enables interrupts and halts.

If any of these steps result in an event handler being invoked, then once the event handler completes, the process starts again at the beginning. This way, hardware interrupts and synthetic events are given priority over repeatedly invoked idle handlers.

As an example, a network card driver is able to implement adaptive polling in the following way: An interrupt is allocated from the `EventManager` and the device is programmed to fire that interrupt when packets are received. The event handler will then process received packets to completion and return to the `EventManager` which will re-enable interrupts. If the interrupt rate exceeds a configurable threshold then the driver disables the interrupt (via programming the device) and installs an `IdleHandler` to process received packets. The `EventManager` will then repeatedly call the idle handler, effectively polling the device for more data. When the packet arrival rate drops below a configurable threshold, the driver re-enables the interrupt and disables the idle handler to return to interrupt-driven execution. While the `EventManager` interface is simple, it provides sufficient functionality to implement this dynamic behavior.

A common challenge associated with non-preemptive programming occurs when a code path must be modified to wait for the completion of an asynchronous event (e.g. write a file). In traditional systems, a thread will block until the event completes and wakes up the thread to continue. However, in non-blocking systems this is not possible. Instead, all calls along the path must pass along a continuation to be invoked when the event completes. Adya et al.[14] refer to this as *stack ripping*. Given our desire to enable reuse of existing software, a purely non-blocking system is infeasible. Much existing software relies on blocking I/O interfaces and modifying these to pass along continuations is often an immense engineering effort. Instead, EbbRT provides a hybrid model that allows events to explicitly save and restore event state (the stack and volatile register state) via the `EventManager`'s `SaveContext` method. This has allowed us to quickly port software libraries which require a blocking system call. At the point where the block would occur, the current event saves its state and processing of pending events proceeds as previously described. The original event state can be restored and resumed via the `EventManager`'s `ActivateContext` method when the asynchronous work completes. The save and restore event mechanisms enable explicit cooperative scheduling between events in order to provide familiar blocking semantics and interfaces.

Finally, non-preemptive execution also provides for a simple implementation of *Read-*

Copy-Update (RCU) [67, 52]. Typically, concurrent data structures are protected by a reader-writer lock which allows either multiple readers or an exclusive writer to access shared data. Reader-writer locks have two primary drawbacks. First, the bookkeeping operations needed to keep track of the number of readers are expensive and may scale poorly. Second, readers are blocked by any writer which may be detrimental for read-heavy workloads. RCU is a software technique which allows wait-free reads of a concurrent data-structure. Simply, readers may execute their critical section without coordination with other readers and writers, so long as read-side critical sections are short. RCU writers must make their updates atomically (typically by swapping a pointer from an old version of the data structure to a newer version), but also. RCU keeps track of *quiescent points* where it can be known that all reader critical sections at some previous time must have completed. Once all active threads have reached a quiescent point, one can be sure that all readers will see the affect of the writer's crtical section.

Consider a linked-list; an exclusive writer can remove an element of the linked list by atomically writing the `next` pointer of the previous element. With wait-free readers, the writer cannot just free the memory of the removed element as readers may be concurrently accessing it. Instead, the writer will wait until the quiescent point following the atomic write at which point no readers can see the previous element. At this point, the writer is allowed to reclaim the memory of the previous element.

Linux provides an implementation of RCU where reader critical sections require disabling preemption and it can be known that a thread cannot be in a reader-side critical section each time it is preempted. This allows readers to avoid coordination at read-time and defers the necessary bookkeeping to the kernel scheduler. A quiescent point is defined to be a point in time when all threads alive since a previous point in time have been preempted.

EbbRT provides an RCU interface via the `EventManager`. System or application software can pass an event handler to be invoked when all currently existing readers can be guaranteed to have completed. As long as readers of RCU protected data structures complete their reads without relinquishing control of the processor (by blocking their event), then we can conservatively say that all readers have completed when an event has been dispatched on every core. Our default `EventManager` uses a simple token-passing algorithm to implement this. Initially the first core holds a token and registers a timer at which point the core passes the token to another core in a ring. Every time a core receives a token, it sets aside all RCU event-handlers registered since the previous token was received. These event handlers can then be invoked on the following reception of the token which indicates that all cores have dispatched an event (at least the timer event) since the RCU event-handlers were registered.

EbbRT Events and Threads: The ISO C++14 standard defines:

> A *thread of execution* (also known as a *thread*) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread.

Note that this definition does not require that threads be preemptive. From a programming language perspective, EbbRT events are a valid implementation of C++ threads. This allows us to map thread interfaces to EbbRT events. For example, thread creation can be implemented by spawning a synthetic event. We ensure that other interactions with the language and EbbRT events match the definition of C++ threads. For example, any C++ exception caught by the `EventManager` when invoking an event handler results in a system abort. This mimics the behavior of C++ calling `std::terminate` when the initial function of a `std::thread` throws an exception. Additionally, "thread local" storage (e.g. `pthread_key_create`) is stored as part of each event's state. Furthermore, a complete implementation of the pthread interfaces could be implemented on top of EbbRT's events — allowing unmodified application code to execute at a low-level context.

3.1.1.1 Advantages

The most obvious advantage of EbbRT's events are how immediately they map to hardware interfaces. An interrupt causes user code to be invoked without the typical overheads of scheduling and context switching. Much of our evaluation demonstrates the benefits provided by this approach. In this section, however, we discuss secondary advantages to this approach.

While EbbRT's events are analogous to threads, they execute without preemption. Many Ebbs use per-core data structures to achieve multi-core scalability. In a preemptive system, accessing per-core data structures would require atomic operations, which can be expensive even if uncontended [32]. In EbbRT, events cannot be preempted and will never be migrated across cores. This allows developers to use non-atomic operations to access per-core data structures. This integrates well with the Ebb model where per-core representatives are used to hold per-core data.

Another advantage of non-preemptive execution is that saving the state of an event is performed via an explicit function call, meaning only callee saved registers and the stack must be saved. Under the x86_64 System V ABI, this only requires saving seven registers (`%rsp`, `%rbp`, `%rbx`, `%r12` through `%r15`). In comparison, preemption requires preserving all register state, including all general purpose registers, floating pointer registers, and flag registers.

Non-preemption helps with the implementation of RCU because no event can be preempted while reading a data-structure. The RCU implementation in Linux, on the other hand must explicitly disable/enable interrupts around RCU access to ensure that preemption does not occur. Userspace implementations of RCU [12] must resort to other techniques like requiring developers to periodically run the RCU subsystem or use `SIGALRM` and explicitly record concurrent reads. As with other operating systems, RCU is used frequently in EbbRT to construct efficient and scalable data structures such as linked lists and resizable hash tables. Furthermore, RCU can used as easily in EbbRT application software as in lower-level system software.

In summary, EbbRT's event-driven execution model provides a number of ancillary benefits such as efficient access to per-core data structures, efficient context-switching, and a natural implementation of RCU.

3.1.1.2 Limitations

In this section we discuss some of the limitations of EbbRT's event-driven execution model. Some of the limitations are an inherent design trade-off necessary to achieve the above advantages. Other limitations, we believe, can be addressed in future work.

A limitation of the non-preemptive model is that it is difficult to map long-running threads with no I/O to an event-driven model. If the processor is not periodically yielded, then event starvation can occur. At present we do not provide a completely satisfactory solution. There are several alternatives we have not yet explored, each with tradeoffs.

1. Building a preemptive scheduler on top of events similar to Scheduler Activations [16] would be possible. However, Ebbs designed for non-preemptive execution (such as those that access per-core data without atomics) would not be usable. Fragmenting the set of Ebbs into those that depend on non-preemptive execution and those that do not is not an attractive proposal given our focus on reducing software maintenance.

2. We could dedicate one or more cores to running each long-running thread, as IX[22] did, and alleviate the need for any preemption. This may be a waste of resources if the long-running thread is periodic.

3. We could also only allow long-running threads on the hosted system. The most significant advantages of the native execution environment come from direct access to hardware. Long-running threads are less likely to see significant advantages by running on EbbRT's native environment.

One implication of the shift from purely asynchronous execution to cooperative multitasking is the increased potential for deadlock. Consider a deadlock we discovered early on in the development process. Our ethernet driver was invoked on a packet receive interrupt. The device will not fire further interrupts until the driver explicitly enables them. The driver iterates over all received packets, passing each to the network stack to be processed synchronously. The network stack eventually invoked an application event handler to handle TCP/IP input. The application code then performed some processing and saved its state to be restored on future TCP/IP input. This caused a deadlock, as the device driver had not re-enabled future interrupts (or processed any other received packets). Fundamentally, the device driver code had an implicit dependency on running to completion (in order to re-enable interrupts) which was violated by the application software saving the event.

The general problem is that synchronously invoking logically independent code causes the invoker to have an execution dependency on the invokee. If the invokee creates a dependency on the invoker (in this case, waiting for additional network traffic), then a deadlock may ensue. This leads to two potential solutions. One solution is to prevent the invokee from creating a dependency on the invoker. Practically, this means restricting the use of the cooperative multitasking primitives to cases where the execution is completely known to create no circular dependencies. This seems challenging in general as the primary motivation for cooperative multitasking is to support porting existing threaded code. For example, it is difficult to know the execution context of a `read` call. And even if it was possible to know that a blocking call could deadlock, there is nothing one can do to resolve the situation. To maintain the synchronous API, the implementation must block.

This leads us to adopt a solution which puts the onus on the invoker to prevent the creation of an execution dependency when the invokee might block. To do this, the invoker should not synchronously call an event handler but instead the event handler should execute on a separate event. This way the invokee is free to create dependencies on the invoker without fear of creating deadlocks.

One of the proclaimed benefits of EbbRT is that application event handlers can execute directly off an interrupt. However, we now may need to create a new event (or several) to eventually execute an application event handler. This may add significant costs to the invocation of an application event handler especially when the expected case is that there is no blocking. Our solution is an implementation of the `Spawn` method which stores the current event's state on a per-core stack and synchronously switches to the newly spawned event before the original event completes. When the new event either completes or blocks, the original event is popped off the per-core stack and resumed. This synchronous `Spawn` prevents us from returning to the event-loop in the common case that event handlers do not block. Coupled with the fact that explicit context switches are cheap, synchronous event spawns provide an efficient method to avoid deadlocks by breaking a single logical event into multiple.

EbbRT's event-driven execution model satisfies two primary goals. First, the primitives provide minimal abstractions in order to support the broadest set of optimizations. Second, this ability to customize doesn't preclude reuse of existing software. Meeting these two objectives allows applications to be ported to EbbRT with little effort and enable significant performance gains to be achieved by allowing application software to execute soon after hardware interrupts. In the next section, we will demonstrate how these two goals are similarly applied to the EbbRT memory model.

3.1.2 Memory

Typical multi-programmed operating systems, utilize a process model. Processes are provided with a large, paged, virtual address space which allows the operating system to transparently multiplex the physical memory resources across multiple isolated program instances. The EbbRT native execution environment does not support multiprogramming and therefore provides a significantly different address space layout. Recall that the primary goal of the EbbRT native execution environment is to provide the flexibility for application developers to exploit application-specific knowledge and specialize software. Therefore, the core address space and memory allocation interfaces offer little abstraction over the hardware capabilities. Rather than provide an interface amenable to multiplexing and isolation, EbbRT's memory interfaces focus on enabling the development of efficient applications and libraries through lightweight abstractions and efficient implementations.

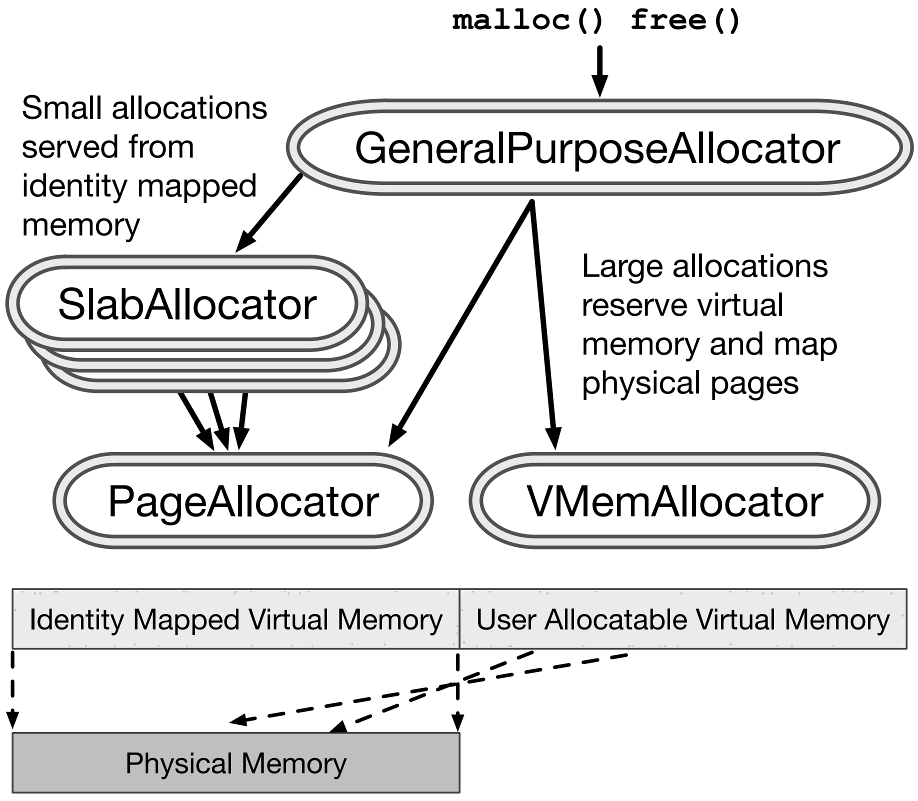

malloc() free()
GeneralPurposeAllocator
Small allocations served from identity mapped memory
SlabAllocator
Large allocations reserve virtual memory and map physical pages
PageAllocator
VMemAllocator
Identity Mapped Virtual Memory
User Allocatable Virtual Memory
Physical Memory

Figure 3.1: EbbRT Address Space and Allocators

Address Space: The EbbRT address space and associated allocators are depicted in figure 3.1. EbbRT provides an address space per-core where the lower-half (47 bits on x86_64) identity maps all available physical memory. A native EbbRT ELF has equivalent link and load addresses so a symbol (code or data) in the binary has a virtual address equivalent to the physical address where it is loaded. This means that all static code and data is accessible from all cores with the same virtual address. Any physical memory that remains beyond application and system code and data is made available via a power-of-two `PageAllocator` Ebb. The `PageAllocator` is NUMA aware — ensuring that pages are by-default allocated from a memory locality closest to the allocating processor. This reduces memory access latency.

The upper-half of the virtual address space is made available at page granularity from the `VMemAllocator` Ebb. Regions of virtual memory can be allocated while providing a user-specified page fault handler. The `VMemAllocator` does not back these virtual regions with physical memory. Rather, it provides utility functions which allow users to establish their own mappings. Each core has its own page table which allows clients of the `VMemAllocator` to provide different mappings for the same virtual address if they wish. In concert with the `PageAllocator`, developers can manage portions of the virtual address space and choose how and when the memory is backed.

The third memory allocator provided by EbbRT is the `GeneralPurposeAllocator` which provides arbitrarily sized allocations of memory. Our modified `newlib` C standard library implements `malloc` and `free` as invocations on the `GeneralPurposeAllocator` Ebb. Users of this Ebb may depend on the allocated (virtual) addresses being mapped to the same physical memory on all cores.

Our default implementation of the `GeneralPurposeAllocator` uses slab allocators [26] which divide regions of memory allocated from the `PageAllocator` into fixed size objects. The size of objects allocated by the slab allocators need not be a power of two. This design is based on the SLQB allocator [31]. The `GeneralPurposeAllocator` maintains a set of slab allocators each responsible for a particular sized object. When an allocation is performed, the `GeneralPurposeAllocator` selects and invokes the slab allocator with the closest object size greater or equal to the requested size. Allocations larger than the largest object size (8 MB by default) are satisfied by allocating a virtual memory region from the `VMemAllocator` and aggressively backing the region with pages from the `PageAllocator`. The individual slab allocators allocate physical pages from the `PageAllocator` Ebb and logically divide them into equal sized objects. The slab allocators return virtual addresses from the lower-half (identity map) which represent the corresponding physical memory. Given this approach, small allocations are served from dense physical memory that is premapped using large pages and larger allocations can sparsely consume physical memory by mapping lazily on page faults. Given the per-core page table structure, large mappings require page-faults on each core that access the allocated memory.

To most EbbRT programmers, the details of the memory subsystem are hidden behind the common `malloc` and `free` interfaces. However, the manner in which these are implemented provides several advantages.

3.1.2.1 Advantages

The EbbRT memory subsystem provides three key advantages. The first advantage is the efficiency gains enabled by using large granularity page mappings. The second advantage is the flexibility enabled by allowing user-allocatable virtual memory regions and mappings. The third advantage we discuss is the synergy between the non-preemptive execution and the per-core memory allocators.

One of the advantages of the design of the memory subsystem is that the lower half of the virtual address space can be mapped with large pages (2 MB or 1 GB). The physical pages themselves may be allocated at smaller granularity and then even further divided as the slab allocators do. Therefore, all static memory (code and data) as well as most dynamic memory allocations are served by large page mapping. An advantage of this approach is that EbbRT applications tend to have smaller TLB footprints as compared to the same applications running on a system which uses small pages by default, such as Linux. Consider that on x86_64 processors, small pages (4 kB) require a four level page-table walk to translate. Each page-table is physically addressed. In a virtualized environment, each page-table address has to be translated from guest-physical to host-physical address which requires an Extended Page Table (EPT) walk. Using larger page mappings ensures page walks are less frequent (due to reduced TLB pressure) and also that such walks are shorter: three levels deep for 2 MB pages and two levels deep for 1 GB pages. As we demonstrate in our evaluation, the efficiency gains by using large mappings are substantial for memory sensitive workloads.

Linux, on the other hand, provides cumbersome and limited support for large pages that typically require application modifications to take advantage of. First, allocating large pages can only be done for runtime memory allocations (e.g. through `mmap`). Due to interactions with the file-cache, executable code and data can not be mapped with large pages. Second, large pages must be explicitly reserved — requiring a system administrator to estimate the portion of memory mappings that can be made with large pages. The shortcomings in large page support is illustrative of the challenges faced by traditional operating systems designed and implemented around supporting multiple processes. In the context of supporting multiple processes concurrently, small pages provides the Linux kernel with the flexibility to share physical memory resources in a balanced fashion. However, this architecture is detrimental for single applications wishing to utilize the full resources of a single machine. EbbRT's memory subsystem is designed and optimized for just such a case.

EbbRT's separation between allocating physical and virtual memory provides developers the flexibility to optionally use smaller pages if desired. For example, the default implementation of the `EventManager` (described in section 3.1.1) allocates event stacks as 8 MB virtual regions and on demand backs the stacks with small, 4 kB pages. In the common case where events are short lived, we expect stacks to be short and not consume the entire 8 MB region and by demand paging the region, we ensure that the physical memory consumption is in accordance with the actual memory consumption. Additionally, the `EventManager` can leave the final page of the region unmapped as a guard against stack overflows. The `EventManager` does not depend on any special interfaces to implement this stack management. The interfaces provided by the `PageAllocator` and `VMemAllocator` are sufficient.

Given the absence of preemption, the `GeneralPurposeAllocator` can service most allocations from a per-core cache without any synchronization. Avoiding atomic operations has proven to be so important that high performance allocators like TCMalloc[38] and jemalloc[35] use per-*thread* caches to do so. These allocators, however, require complicated algorithms to balance the caching across a potentially dynamic set of threads. Given EbbRT's execution model which targets a single active event per-core, the default allocator's implementation can be simplified given that the number of cores is typically static and generally not too large.

Additionally, the strict split between process level memory management and kernel physical page management in a tradional operating system can lead to inefficienies due to the semantic gap. For example, Facebook discovered that running applications using TCMalloc for long periods of time caused the system to begin paging out memory despite the application not increasing its memory usage[34]. The culprit was TCMalloc's per-thread caching. From the application's perspective, memory had been freed, however from the kernel's perspective, the memory was still in use because the memory was cached in the user-level memory allocator and had not been freed to the kernel. The root cause is that the allocator lacks sufficient knowledge to know how much it should cache. In EbbRT, the `PageAllocator` knows when the availability of physical memory is low. We could provide an interface where clients of the `PageAllocator` (e.g. slab allocators) could be notified to flush cached pages. We have not yet explored this, though it remains an option.

3.1.2.2 Limitations

In this section we describe some of the limitations of the EbbRT memory subsystem. In particular we discuss the drawbacks to forgoing memory protection to implement both system and application functionality. We also discuss some of the challenges of allowing applications to handle page faults in a non-preemptive system.

The most apparent limitation to the EbbRT native memory subsystem is that there is a large, shared address space. Aside from user-managed regions of the virtual address space, all code and data are accessible at all times and all code executes at the highest privilege level. This means that malicious code can easily exert the full authority of the machine (talk to devices, read and write arbitrary memory, etc.). Additionally, there is limited fault isolation, a bug in application code can cause cascading errors which only appear as errors in, say, the network device driver.

Memory isolation has been a hallmark of operating systems functionality for decades in order to safely share the resources of a machine amongst multiple users and applications. In EbbRT we relax this requirement by depending on IaaS isolation to multiplex the resources of a data center. Nonetheless, memory isolation is a key method in constructing high-integrity systems. This is one of the core tenets of the microkernel vs monolithic kernel debate. Proponents of microkernels claim that by reducing the authority of system components, we can build more secure and more correct operating systems [58].

While EbbRT does not enforce memory protection, it can restrict the impact an application can have. We can reduce the threat of malicious code by only deploying EbbRT systems on private networks. This limits the system's exposure to potentially malicious inputs and therefore malicious code. We argue that this is a reasonable tradeoff given the domain EbbRT targets. Most cloud applications are composed of services such as in-memory data caching or data storage which have high-performance I/O requirements but are often not publicly accessible. Additionally, most public user requests are front-ended by load balancers which may filter inputs before forwarding to EbbRT system backends.

In keeping with the goal of EbbRT's execution environment being a foundation for customization, one could exploit the existing execution environment to support more restricted sandboxes where user code can be executed with lower privileges. In particular, it would be possible to construct a ring 3 userspace on top of EbbRT with restricted page mappings to provide isolation. In general, these sandboxes cannot simply use those Ebbs which execute under the assumption of full privilege. We have not yet explored the design of interfaces to support the construction of sandboxes with EbbRT.

A significant source of frustration as a developer of EbbRT is the lack of fault isolation. On more than one occasion an afternoon has been spent tracking down a bug to its root cause. The combination of the use of a programming language which doesn't provide memory safety (C++) and a single address space makes it easy to create bugs which have a long-distance effect. We do mitigate these issues in some simple ways. All code is marked as read-only memory and all other memory is marked as no-execute. This is a strategy employed by many OSs, commonly referred to as W^X (Write exclusive-or Execute). The `PageAllocator` could remove freed pages from the address space to catch references to freed memory. We do not currently provide this due to the potential overhead but it could be implemented and enabled only during debug runs.

A more substantial way to address this is the use a memory-safe language. As an example, MirageOS [62] is a framework for building OCaml library operating systems. OCaml is a garbage-collected language which prevents access to arbitrary memory. The major down-side of this approach is that the OCaml runtime has additional overheads, particularly when latency is a concern (due to garbage collection and entering the runtime off an interrupt). A promising alternative, then, is a language which promises memory safety with native performance. Two languages which meet these requirements are Rust [65] and ATS [29].

Providing user allocatable virtual memory and user managed page faults causes some challenges with our non-preemptive execution model. Figure 3.2 illustrates a bug we discovered early in the development of EbbRT: The `EventManager` allocates a 8 MB region of virtual memory from the `VMemAllocator` for each event stack. A deadlock could be triggered in the following case:

1. Application code invokes the `PageAllocator`
2. The `PageAllocator` acquires a mutex (for per-NUMA node data)
3. Before releasing the lock, a page fault occurs due a stack access beyond a page boundary
4. The registered page fault handler executes on a separate exception stack and invokes the page allocator in order to resolve the page fault
5. The `PageAllocator` attempts to lock the already held mutex

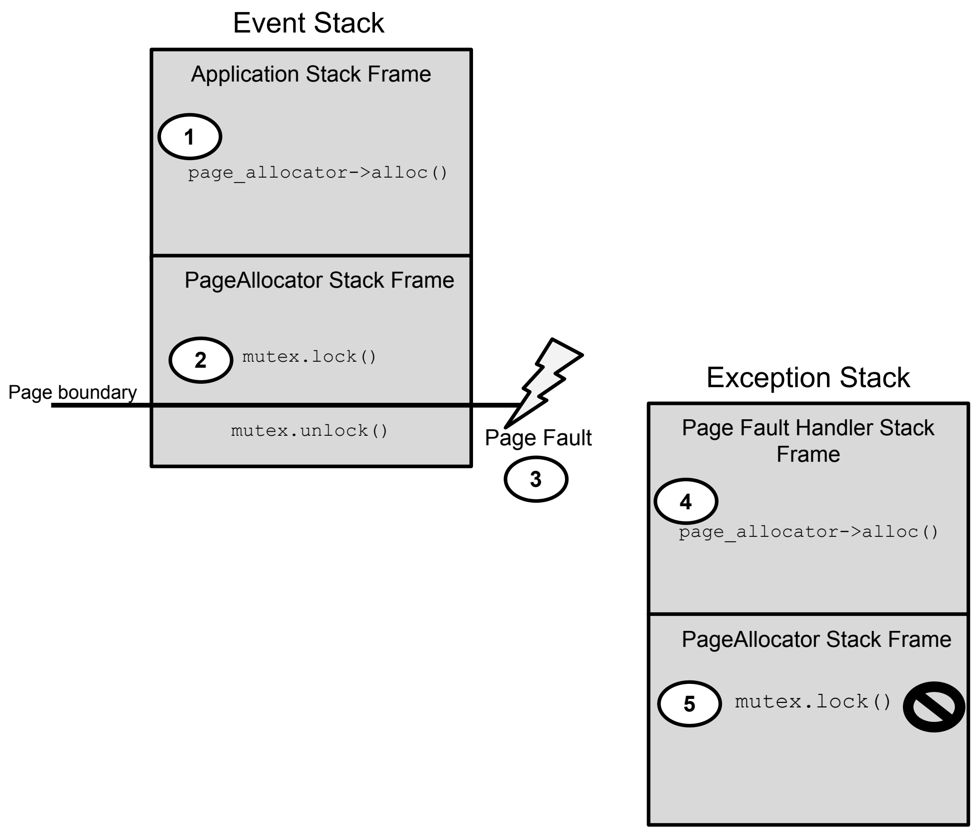


Figure 3.2: Deadlock when resolving page fault

The root cause of this issue is that an event's stack could page fault while executing arbitrary code. So the page fault handler must ensure that all code it executes is re-entrant. Most code in EbbRT is not re-entrant due to the common use of per-core data coupled with a lack of preemption. In this case, we resolved the deadlock by modifying the `PageAllocator` to probe the necessary stack space before acquiring the lock. This caused any stack faults to occur before holding a lock.

This deadlock highlights a more general concern with the `VMemAllocator` and its clients. Any code which can cause a page fault must have some relationship with the code which handles the page fault. For example, arbitrary EbbRT code assumes that the processor can only be yielded explicitly, so a page fault handler which tries to block the faulting event may cause an error. The result of this discovery is that either page fault handlers must make conservative assumptions about the state of an event when it page faults as in the case of the `EventManager`, or the code that can touch faulting virtual memory regions must be written under the assumptions of what its page fault handler may do (e.g. block the event).

Therefore, it would be difficult to generally support interfaces like `mmap` in a manner which allows file data to be faulted into memory on demand since this may require blocking an event (e.g. to read from disk). A generally usable `mmap` interface must aggressively map in file data before returning. To date, this limitation has not prevented us from porting applications, though we have not yet explored those applications which make heavy use of such interfaces.

3.1.3 I/O

The third major component of any execution environment is the manner in which I/O is performed. Most operating systems provide a system call interface which allows applications to invoke privileged routines to perform I/O on their behalf. Common examples include UNIX files (`read`/`write`) and BSD sockets. One issue commonly associated with these interfaces is that they require the system to copy data from kernel memory to application memory on receive and vice-versa for transmit. Another concern is the performance overheads associated with the context switches between kernel and userspace, especially for low-latency workloads.

The primary issue with the POSIX `read` and `write` interfaces is that the application dictates the memory location of the data by providing a pointer. Alloc Stream [53] presents an interface where the application receives a location in memory which is filled for reading or writing. This allows the system to provide a pointer to memory which maps to the in-memory file cache. For network sockets, this could provide memory which could be mapped to memory amenable to DMA for transmit or, for reading, map to the in-kernel packet memory. Copies can be avoided, though there is the drawback of additional virtual memory operations (mappings, unmappings, and TLB operations). IX [22], a system designed for high performance networking applications, similarly provides zero-copy by dedicating a SR-IOV (hardware virtualized) network device to an application. The application then has a pool of buffers which can be used for packet reception and transmission. Because the device is dedicated, the device can DMA directly into the application buffers. One drawback is the fixed pool of buffers which must be released to receive more data. This is necessary to avoid virtual memory operations. If an application wishes to have access to packet data for a substantial period of time, the application must copy the data in order to release DMA buffers.

The EbbRT interfaces to access compute and memory resources are designed to provide the ability to construct high-performance I/O paths. To this end, the system does not mandate how I/O resources are accessed. A software component such as a device driver can access device MMIO by mapping a virtual region allocated from the `VMemAllocator` to the proper physical address. Equivalently, on x86, because all EbbRT software executes in ring 0, a software component can directly access the I/O address space through the `in` and `out` instructions. These software components can directly allocate physical memory from the `PageAllocator` for device DMA access. The `EventManager` provides an interface to allocate an interrupt number (32–255 on x86) and install an event handler to be invoked when that interrupt fires. Alternatively, a device driver can install an `IdleHandler` to directly poll a device.

Device drivers are simply implemented as Ebbs with no additional privileges over any other. Once they execute, they are free to invoke other Ebbs, or spawn additional events, etc. This gives developers the flexibility to provide a wide range of I/O interfaces with a variety of implementations.

It should be noted that writing high performance I/O paths is difficult. We do not expect all application developers to do so. EbbRT is designed to enable the encapsulation and reuse of I/O components that have been written to take advantage of the features of EbbRT's execution environment: Supervisor level privilege, control of physical and virtual memory, and direct and flexible control of interrupt handling.

3.1.3.1 Advantages

The default EbbRT network stack highlights several advantages of this approach to I/O. EbbRT includes a custom network stack for the native environment providing IPv4, UDP/TCP, and DHCP functionality. The network stack is designed to provide an event-driven interface to applications and minimize multi-core synchronization while enabling pervasive zero-copy. The network stack does not provide a standard BSD socket interface, but rather enables tighter integration with the application to manage the resources of a network connection. Work to implement a socket interface on top of our existing network interface is ongoing.

The network device driver is driven either by a device interrupt or by polling (as described in Section 3.1.1). For each packet that is received, an event is synchronously created and invoked to perform network processing. Ethernet, IPv4 and UDP/TCP processing is performed until, finally, data is provided to the application by invoking a pre-installed handler. The network stack does not provide any buffering, it will invoke the application as long as data arrives. Sending data is similarly synchronous; an application invokes the network stack which processes until a packet is sent by the network card.

Most operating systems have buffers in the kernel which are used to pace connections (e.g. manage TCP window size, cause UDP drops). In contrast, EbbRT allows the application to directly manage its own buffering. In the case of UDP, an overwhelmed application may have to drop datagrams. For a TCP connection, an application can explicitly set the window size to prevent further sends from the remote host. Applications must also check that outgoing TCP data fits within the currently advertised sender window before telling the network stack to send it or buffer it otherwise. This allows the application to decide whether or not to delay sending to aggregate multiple sends into a single TCP segment. Other systems typically accomplish this using Nagle's algorithm which is often associated with poor latency [68]. An advantage of EbbRT's approach to networking is the degree to which an application can tune the behavior of its connections at runtime. We provide default behaviors which can be inherited from for those applications which do not require this degree of customization.

One challenge with high-performance networking is the need to synchronize when accessing connection state [74]. EbbRT stores connection state in an RCU [66] hash table which allows common connection lookup operations to proceed without any atomic operations. Connection state is only manipulated on a single core which is chosen by the application when the connection is established. Therefore, common case network operations require no synchronization.

The EbbRT memory system enables pervasive zero-copy throughout the network stack by providing the ability to allocate physical frames from the `PageAllocator`. Additionally, the non-preemptive event-driven execution environment allows the network stack to synchronously process packets from an interrupt up to the application and back without additional context switches or address space changes. This combination enables significant performance advantages for applications requiring high-throughput and/or low-latency I/O. We demonstrate these advantages in our evaluation.

The EbbRT network stack is an example of the degree of performance specialization our design enables. Because of the lack of buffering in our network stack, the application can be involved in network resource management. This also reduces complexity in our network stack, as much of the complex policy decisions (sizing of buffers, management of TCP windows) can be made by the application. Historically, network stack buffering and queuing has been a significant factor in network performance. EbbRT's design does not solve these problems, but instead enables applications to more directly control these properties and customize the system to their characteristics. The zero-copy optimization illustrates the value of having all physical memory identity mapped, unpaged, and within a single address space.

3.1.3.2 Limitations

The I/O interfaces offer an extreme amount of flexibility by effectively providing no abstraction over the hardware interfaces (e.g. MMIO, DMA, `in/out`). However, this flexibility is in some manner limited by the desire to provide and use higher-level interfaces. For example, we provide a network stack interface. While the implementation could be replaced, the interface is harder to change because other Ebbs have and will continue to depend on it. One must consider then, how the in-use I/O interfaces restrict the flexibility provided by the execution environment. Interfaces inherently restrict implementations and each defined interface has inertia which makes it difficult to modify.

Currently, we strive to ensure that the I/O interfaces that we provide are structured such that there would be marginal benefit in bypassing them. We provide low-level interfaces where possible and construct higher-level, more standard interfaces on top (e.g. sockets). With the low-overhead of Ebb interfaces and our whole-program compilation model, such a layering approach has little-to-no performance overhead. Particularly for the network stack, we use a layered approach to enable developers to customize functionality as they deem necessary (e.g. implement their own buffering logic). This structuring alleviates our concern of the interfaces ossifying because new implementations can be built on top of whichever layer a developer chooses without requiring underlying implementations to change.

3.1.4 Summary

The native execution environment provides interfaces to manage compute, memory and I/O resources. Our focus on low-level interfaces with little abstraction enables software specialization for high performance. On the other hand, our interfaces provide enough flexibility to be suited for a wide range of applications.

3.2 Hosted Environment

The EbbRT hosted execution environment looks similar to the native execution environment — it supports Ebbs and an event-driven execution model. This similarity is a consequence of our desire to reduce the need to create and maintain non-performance critical code rather than an explicit goal. We provide an analagous `EventManager` Ebb with many of the same interfaces like `Spawn`. However, we do not provide an analagous memory system. This would require maintaining additional software for little to no gain over merely using the system `malloc` implementation.

The hosted environment allows developers to quickly integrate legacy compatible software into their EbbRT applications. Rather than porting large amounts of non-performance-critical software (e.g. POSIX compatible filesystems for configuration or logging libraries) to the native execution environment, developers can offload functionality to a hosted execution environment which runs within a process of a general purpose operating system such as Linux. Likewise, a large Linux application can integrate the hosted execution environment to offload performance critical functionality to the native environment.

The hosted execution environment must be extensible — developers should be able to add, modify, or remove support for legacy functionality as their application dictates. Note that the need for extensibility within the hosted execution environment mirrors the need for customizability in the native execution environment. The Ebb model provides a unifying abstraction which spans both execution environments.

Additionally, the hosted execution environment should allow itself to be usable in a wide variety of scenarios to broaden its applicability. To this end, we provide the hosted execution environment as a library which can be linked into existing applications. Furthermore, the execution model is event-driven rather than multi-threaded. This allows it to be embedded into either event-driven applications (as an event) or multi-threaded applications (as a separate thread).

It should also be noted that the hosted environment is not a requirement for an

EbbRT application. For example, our implementation of memcached provides a protocol-compatible network interface from which clients (Linux or otherwise) can interact with the EbbRT application. Additionally, an EbbRT application may provide both a network interface as well as rely on a hosted environment. Our implementation of a node.js webserver provides a standard HTTP interface, but also relies on a hosted implementation for file I/O.

3.3 Conclusion

Together the EbbRT execution environments enable developers to construct performance-optimized applications without sacrificing reusability, common libraries, or existing operating system interfaces and implementations. In our native execution environment, we adopted an event-driven execution model as the abstraction for compute resources which enables applications to execute directly off of hardware interrupts. We also allocate physical and virtual memory resources separately to similarly allow applications more control over memory resources. Finally, I/O access is not prescribed by EbbRT and allows application-specific interaction with device hardware. Finally, the hosted execution environment supports embedding EbbRT functionality into existing applications by adopting an event-driven execution environment and similar programming primitives to the native environment.

Chapter 4

Modularity

In the previous chapter, we introduced the EbbRT execution environment which describes the core primitives which define the application interface to hardware resources. In order to expose the raw features and performance of the hardware, our primitives tend to be at a lower level of abstraction that a typical operating system. For example, UNIX operating systems provide file abstractions upon which operations such as reads and writes can be performed. While there are other UNIX primitives, almost all resources supporting by the kernel, both real hardware devices and logical entities, are expected to expose a file-based interface that hide resource-specific details and features. In constrast, EbbRT does not provide such higher-level primitives. Instead the base execution environment simply provides, as discussed, interfaces for efficient per-core event-driven execution, memory management, and raw I/O device operationg and data transfer. The downside of using low-level interfaces is that programming can often be more verbose and time-consuming. Furthermore, existing applications and libraries would have to be ported to target our new interfaces. We believe that this effort is only worthwhile for performance-critical portions of an application and so we strive to also support higher-level interfaces without preventing access to the lower-level interfaces.

Our goal is to support higher-level programming interfaces and environments while providing some method in which an application can selectively use lower-level interfaces as needed. EbbRT applications, therefore, are constructed in a modular fashion, where software components are recursively built of smaller components which can be individually used to construct more specialized implementations. Modularity is an important property of EbbRT and deeply influences the design and implementation of EbbRT.

There are several aspects to pursuing modularity which must be considered:

1. There must be some mechanism by which an interface is defined such that the particular details of an implementation are hidden from a client. This is classically referred to as information hiding [72]. Information hiding allows software components to be more freely changed in isolation from the rest of the system.

2. Interfaces can often restrict implementations. Consider the previously discussed POSIX `read` and `write` interface. The interface makes zero-copy implementations of I/O impossible. A different interface, like Alloc Stream [53] enables zero-copy implementations without restriction. We refer to this as the *flexibility* of an interface. The interfaces should be flexible enough to support a diverse set of implementations (otherwise the distinction between interface and implementation is moot).

3. To encourage fine-grained decomposition, one would like to minimize overheads incurred by modularity (in particular, interface crossings) without concern that the functionality is "too small" to justify the overheads associated with interface crossings. Otherwise, developers, seeking performance, will be encouraged to create large modules which, by their nature, are more difficult to modify or replace than smaller ones.

4. Modularity by itself is insufficient for enabling customization and reuse. Many operating systems can be described as modular in the sense that software components have defined interfaces and can be developed independently. However, two modules can have interdependencies which make it impractical to use one without other. For example, Linux device drivers are clearly modular, but their dependency on all the interfaces Linux provides makes it difficult for one to reuse a Linux device driver in a different system. The OSKit[37] authors refer to this as the difference between *modularity* and *separability*. A collection of components which cannot be independently

used (and therefore, lack separability), is, from the perspective of customizability, one large component which may be difficult to modify.

In this chapter we present EbbRT's approach to modularity in two sections. First, we describe Elastic Building Blocks, the mechanism for defining modules and their interfaces and invoking them. Next we consider primitives which cross module boundaries and how they are used in the definition of module interfaces.

4.1 Elastic Building Blocks

EbbRT adopts an object model, Elastic Building Blocks. As with objects in many programming languages, they encapsulate implementation details behind an interface. The namespace of Ebbs is shared across all machines in the system (hosted and native). This means that any Ebb can be referenced from any machine on the system (as opposed to referenced only on a local machine). Ebbs are distributed, multi-core fragmented objects [27, 64, 75]. When an Ebb is invoked, a local *representative* handles the call. Representatives may communicate with each other to satisfy the invocation. For example, an object providing file access might have representatives on a native instance simply function-ship requests to a hosted representative which translates these requests into requests on the local file system. By encapsulating the distributed nature of the object, optimizations such as the use of RDMA, caching, and local storage, etc. are all be hidden from clients of the filesystem Ebb.

Many programming languages such as C++ exploit custom calling conventions in the construction and implementation of interfaces. As such, methods of C++ objects cannot be invoked from other languages or address spaces due to the lack of well-specified binary interfaces. Object models like COM [81] and CORBA [79] define language-agnostic, binary interfaces for objects in order to enable language-agnostic or cross-address space invocation. In doing so, every client must serialize data structures for crossing an object boundary. Therefore local invocations of objects look identical to invocations of objects on remote systems or within other address spaces. However, this unnecessarily increases the cost of all local object invocation.

An alternative approach is to define a binary interface using a data interchange format like Protobufs [40] or Capnproto [3] and explicitly message (typically remote) endpoints. Similar to the distributed object models, this requires all invocations to behave as if the endpoint is remote and does not encapsulate efficient local optimizations like caching.

Developers define an Ebb's functional interface using a standard C++ class definition. We refer to this as the *Ebb class*. Each instance of an Ebb is associated with a system wide unique identifier, called an `EbbId` which is obtained from a system object called the `EbbAllocator` when an Ebb is instantiated. Invocations on an instance are done through a reference type, called an `EbbRef`, that is used to translate an instance's `EbbId` to a class-specific representative that will handle the invocation locally on the machine and core that the call was made. The conversion of an `EbbId` to an `EbbRef` can differe between the native and hosted environment. Therefore, `EbbIds` are a global way of identifying an instance and can be exchanged between machines. Hwoever, `EbbRefs` are machine specific.

In this section we first discuss the mechanics of the Elastic Building Block primitive (in particular, invocation, fault handling and communication) and then have a longer discussion about the outcomes of our design.

4.1.1 Invocation

Ebbs may be invoked on any machine or core within the application by passing around the `EbbId` and then casting it to an `EbbRef`. An `EbbRef` can be dereferenced to a local (generally per-core) representative. Therefore, all invocations of an Ebb are handled by the local representative. Any communication between representatives (whether on separate cores or separate machines), is encapsulated from the client. One of the benefits of this approach is that clients do not need to serialize parameters when invoking an Ebb, it is the responsibility of the Ebb to do so internally if the implementation requires it.

The EbbRT native execution environment reserves a portion of the virtual address space for use by the Ebb subsystem. In particular, each `EbbId` is an offset into this virtual memory region which contains a reference to the per-core *representative.* We refer to this region as the *translation memory.* This region is large (32 GB of virtual memory) and quite sparse. An `EbbRef` is simply an address in the translation memory which holds a pointer to the per-core representative. Therefore, invoking an Ebb incurs only a single dependent pointer dereference in addition to a standard C++ object invocation.

It is desirable for initialization of the per-core representatives to happen on-demand. In the case that an Ebb is short-lived and only accessed on one core, initializing representatives aggressively on every machine in the system would incur significant overhead. In order to accommodate on-demand initialization, we consider the translation memory as a cache for the per-core representatives. Under memory pressure, the system may evict a page.

The caching mechanism works as follows: When an EbbRef is dereferenced, if a non-null value is stored in the corresponding entry of the translation memory, then that value is a reference to the per-core representative. Otherwise, a type-specific *fault handler* is invoked which must return a reference to a representative or throw a language-level exception. Typically, a fault handler will construct a representative and store it in the corresponding translation entry so future invocations will take the fast-path.

Figure 4.1 shows a subset of the `EbbRef` implementation. Line 2 shows that an `EbbRef` is simply a word-sized value which references the translation memory. This can be constructed from an EbbId by taking the EbbId as an offset into the translation memory (as seen on line 4). Line 10 shows the check to ensure a reference is cached in the translation memory, and if not the class specific fault-handler is invoked on line 13.

Treating the translation memory as a cache integrates nicely with the desire to initialize per-core representatives on demand. An implementation supporting paging can evict a page of the translation memory and need not persist the page. Rather, on a future access to the page, it can be backed with a zero page which will cause all invocations to fault again. Therefore, the fault handler is responsible for locating a reference to a per-core representative (if one exists) or constructing one.

```
template <class EbbClass> class EbbRef {
    uintptr_t ref_ /* Reference into translation memory */
  public:
    constexpr explicit EbbRef(EbbId id = 0)
      : ref_(trans::kVMemStart + sizeof(uintptr_t) * id) {}

    EbbClass& operator*() const { /* Dereference operator */
      EbbClass *lref = *reinterpret_cast<EbbClass**>(ref_);
      /* Check if a reference has not been cached */
      if (unlikely(lref == nullptr)) {
        EbbId id = (ref_ - trans::kVMemStart) / sizeof(LocalEntry);
        /* Invoke class-specific fault handler */
        lref = &(EbbClass::HandleFault(id));
      }
      return *lref; /* Dereference to return EbbClass reference */
    }
}
```

Figure 4.1: EbbRef implementation and dereference operator

We implement an `EbbRef` as a C++ class which overloads the dereference operator. The `EbbRef` class is templated by an Ebb's functional interface (a C++ class). An `EbbRef` is created by casting from an `EbbId`. On the native environment an `EbbRef` stores one value, the address within the translation memory which corresponds to the `EbbId` from which the `EbbRef` was constructed. When an `EbbRef` is dereferenced, it loads a word from this address; if it is non-null, then it is treated as a reference to a representative (in particular, a reference to the template parameter) and returned by the dereference operator. Otherwise, the fault handler is invoked.

On the hosted environment, we do not have access to per-core virtual memory to implement the translation memory as we do on the native environment. Rather, we use a per-thread hash table (keyed by `EbbId`s). An `EbbRef` dereference performs a hash table lookup to find the representative or, if none is found, invoke the fault handler. Thus, thee overheads of an Ebb invocation is higher in the hosted environment.

4.1.2 Fault Handling

The static `HandleFault` method of the Ebb class is invoked with the corresponding `EbbId` passed as a parameter. This function returns a reference to a per-core representative which can then be invoked. The `HandleFault` method can be defined by the Ebb developer or inherited from base classes which provide common behavior.

To construct a representative may require data or communication with other representatives. EbbRT defines several Ebbs (and provides default implementations) in order to resolve faults. First, the `LocalIdMap` provides concurrent access to per-`EbbId` data within a machine. This is simply a shared memory hash table keyed by an `EbbId`. Concretely, the `LocalIdMap` allows locked read and write access to individual entries based on the `concurrent_hash_map` from Intel's Threading Building Block library [11]. The value type is `boost::any`, a type which can hold values of arbitrary types. A common use of the `LocalIdMap` is to persistently store references to per-core representatives. Recall that the translation memory is merely a cache for per-core representative references. An Ebb can store a hash table within its `LocalIdMap` entry. This hash table stores references to representatives keyed by the core they were constructed for. When a fault occurs, the fault handler will first check the `LocalIdMap` to see if the representative has already been constructed.

Another Ebb, the `GlobalIdMap` provides system-wide distributed per-`EbbId` storage. The current interface provides key-value `get` and `set` operations where keys are `EbbIds` and values are arbitrary binary data. Given it's distributed nature, we must define the semantics around concurrent access and behavior in the face of failures. We define these operations to have the strongest achievable form of distributed consistency, *linearizability* [44]. That is, the result of all operations is the same as if they were reordered to occur at a particular point on a common (global) timeline subject to the constraint that if the response of an operation (e.g. a `get`) preceded an invocation of another operation (e.g. a `set`) in the original history, then the first operation still precedes the second in the reordering. This strong consistency provides the maximal flexibility to the programmer (those with loose consistency requirements can still use the `GlobalIdMap`). However, this comes with the added overhead required by consensus protocols. The current version of the `GlobalIdMap` provides a key-value store interface. We expect this to eventually provide a richer interface such as Zookeeper [45] in order to better support group services such as synchronization.

With these `IdMaps`, Ebb creation typically proceeds as follows:

1. Allocate an `EbbId` from the `EbbAllocator`. This reserves a system-wide unique `EbbId` and also implicitly reserves an entry in each of the `IdMap` Ebbs.
2. Write necessary configuration to the `GlobalIdMap` in order to later resolve faults.
3. Upon completion of the write to the `GlobalIdMap`, the `EbbId` can be passed around, and converted to an `EbbRef` and invoked.

Any future fault should then be able to retrieve the configuration from the `GlobalIdMap`. Note that in this scenario, linearizability is necessary. Because we do not track the communication between machines (and in particular, the passing of `EbbIds`), linearizable operations are the only way to ensure that a future read of the `GlobalIdMap` (on any machine) will see a previous write.

While we provide both these Ebbs in a standard deployment of EbbRT, developers are free to create others and use them to satisfy a fault. The `IdMap` Ebbs are merely a base upon which other Ebbs can be constructed; the system does not require they be used by every Ebb. Developers can, for example, provide distributed data store Ebbs with weaker consistency but higher performance and use that in lieu of (or in addition to) the `GlobalIdMap`.

4.1.3 Communication

As all Ebb invocations are inherently local, distributed functionality is provided by representatives communicating among themselves. Here, again the `IdMap` Ebbs may be used just as they could be during construction and fault handling. For example, shared data may be stored in the `GlobalIdMap` and accessed by any representative. In addition to this, EbbRT provides a `Messenger` Ebb which provides point-to-point unidirectional messaging. Clients specify a `NetworkId` (under our current implementation this is an `IPV4` address) and a standardized EbbRT buffer object (called an `IOBuf`, discussed in section 4.2.2) containing data to be sent. The `Messenger` will ensure that if the host is available (e.g. the receiver has not crashed and is not partitioned from the sender), then some representative corresponding to the invoking Ebb will eventually be invoked on the receiving host. Ebb classes desiring this functionality should inherit from the `Messagable` class which requires the implementation of a `ReceiveMessage` method which takes two arguments, the sending `NetworkId` and an `IOBuf` containing the message contents. The `Messenger` ensures that all messages between two hosts are delivered in FIFO order. Under our current implementation we use TCP to guarantee both order and lack of corruption.

One interesting aspect of the `Messenger` Ebb is how it resolves representatives on a remote host. Recall that Ebbs are not instantiated aggressively on all machines in the system, so it is often the case that the `Messenger` Ebb will cause the target Ebb's fault handler to be invoked. The challenge here is that the fault handler is a static method of the Ebb Class, and so the `Messenger` must be able to dynamically resolve the type of the target Ebb. We achieve this through inheritance from the `Messagable` class which causes all messagable Ebbs to register their class at boot-time which creates a mapping from a type-specific hash-code (defined for all C++ types) to a function which resolves an EbbId to the given type. The `Messenger` Ebb prefixes each message with a header which specifies the EbbId of the target Ebb as well as the hash-code of the type. This mechanism enables the `Messenger` to fault in a representative if needed.

Messages contain arbitrary binary data and Ebbs are free to use their own layout. EbbRT supports Capnproto [3], a data interchange format and associated C++ stub compiler and library which allows one to create rich data objects (containing lists, maps, text, integers, etc.) with a well-defined serialized binary format. Capnproto is similar to JSON (but binary) and Protobufs [40]. The key difference between Capnproto and Protobufs is that the in-memory format of a Capnproto object is naturally serialized. This means that there is no additional overhead when an object needs to be written out to the network, or vice versa, when an object is read from the network. This integrates nicely with the EbbRT networking and messaging subsystems which support zero-copy. As with many aspects regarding Ebbs, developers are free to use their own preferred communication protocol, there is no requirement to use Capnproto or the `Messenger` itself.

4.1.4 Advantages

The three key advantages that Ebbs provide are: 1. Location transparency 2. Low overhead invocations 3. Flexible implementations. In this section, we discuss these advantages in turn.

The Elastic Building Block model embraces location transparency where, from a client's perspective, there is no distinction between local and remote operations. This synergizes well with our distributed architecture; an Ebb can provide an interface which appears to it's client as a local operation (e.g. a filesystem interface) yet a native representative may function-ship to a hosted representative and use the filesystem on an existing system such as Linux. The filesystem Ebb can also be replaced with an implementation which uses local storage, caching or any number of different techniques without requiring changes to the clients. Developers can build simple, initial implementations of Ebbs and optimize them later as profiling dictates. This is critical to support our goal of reducing development effort.

In addition to location transparency, even the lowest levels of the native environment are modular; Ebbs have lightweight run-time requirements and nearly zero-overhead. For example, we define the native memory allocator as an Ebb (using per-core representatives for locality) so that it can be easily replaced by a developer. Concretely, the fast-path Ebb invocation requires a memory dereference (to load the word stored in translation memory) and a predictable conditional branch (to check if the fault handler needs to be invoked first) on top of the standard C++ object invocation. These overheads are quantified in chapter 6.

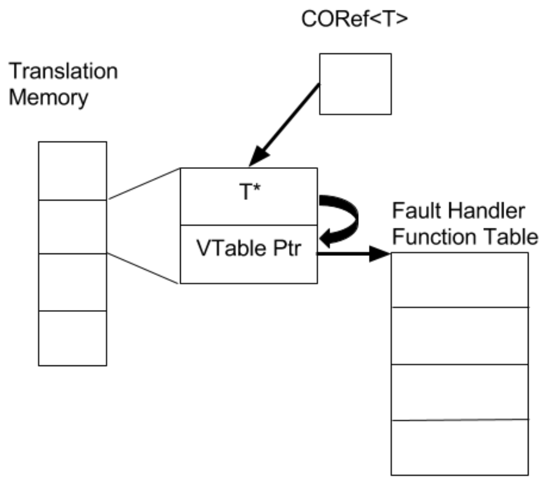


Figure 4.2: Clustered Objects Invocation

Another low-overhead fragmented object-model, Clustered Objects [18], is quite similar in its implementation. Figure 4.2 illustrates the invocation of a `CORef`. Cluster Objects similarly have an area of per-core virtual memory. Unlike Ebbs, Clustered Objects rely on the common implementation of C++ virtual functions where the first word of an object is a pointer to a table of function pointers which can be invoked dynamically. Clustered Objects, like Ebbs, have a fault-handling mechanism which is implemented by initially replacing the virtual function table pointer (stored in the second word of each translation memory entry) with a default, capable of resolving faults. Therefore, rather than an explicit conditional to check if a Clustered Object is initialized, calls go through the indirection provided by the virtual function table which may interpose on the invocation to instantiate a representative. One of the disadvantages of the approach taken by Clustered Objects is that methods must be virtual. Ebbs can use non-virtual methods which has the advantage that at a call-site, the compiler can know the concrete implementation of the called method. Coupled with the fact that we use link-time optimization to enable the compiler to optimize across the whole binary, this can provide significant performance gains. As an example, the default EbbRT memory allocator is implemented as a collection of fixed-size memory allocators (also implemented as Ebbs). Each allocation request is rounded up to the nearest fixed size and then forwarded to the corresponding allocator. Most memory allocations are of a fixed size (e.g. the C++ `new` operator allocates a fixed size object). In such cases, we witnessed the compiler would inline the implementation of the default allocator and directly invoke the corresponding fixed size allocator. This was only possible due to the use of static function dispatch (the implementation was known at the call-site) and whole-program optimization. Note that this static specialization does not prevent the construction of polymorphic Ebbs. One can construct an abstract Ebb class with pure virtual methods. A concrete implementation would, at construction time, store its fault handler in the `LocalIdMap`. The abstract class's fault handler then simply forwards the fault to the fault handler stored in the `LocalIdMap`.

In designing Ebbs, we strived to provide as much flexibility as possible. For example, fault handling is left to be implemented on a per-Ebb class basis using any means a developer wishes. For example, we provide the `LocalIdMap` and `GlobalIdMap` but developers are under no pressure to use them to implement fault handling. This allows us to construct these utility Ebbs without concerning ourselves with providing an ideal solution in all cases. Similarly, we provide the `Messenger` Ebb to enable inter-represntative communication, but there is nothing first-class about this Ebb, developers can construct or use any Ebb for communication.

4.1.5 Disadvantages

Waldo et al. [49] criticize the approach of location transparency as hiding information necessary for the client. In particular they list latency, memory access, partial failure, and concurrency as key differences between local and remote operations which clients must be aware of.

The first difference, latency, is perhaps the most obvious. The authors claim that ignoring the latency differences between local and remote operations can lead to designs with performance problems due to a large amount of communication. Before we can address this claim, consider the latency differences of various operations listed in Table 4.1. All CPU operations are based on the Intel Haswell microarchitecture. A minimal function call and return takes about 3 ns whereas a TCP round trip can take anywhere from 100 000 ns to 30 000 ns depending on the environment. This represents roughly a 10000 fold latency increase for a remote operation. However, because Ebbs encapsulate their communication, the system does not force the use of a particular communication paradigm like TCP. Should an Ebb be capable of using RDMA over Infiniband, operations can drop to about 1500 ns, only a 500 fold increase over a local operation. This is still quite a significant difference. But consider that the overhead of an in-core page fault is roughly 500 times the overhead of a load from the L1 cache. Yet only in extreme cases do programmers concern themselves with pinning memory to guarantee page faults can't occur when referencing memory. In fact, far more common is to solve performance issues (caused by excessive page faults or otherwise) by profiling an application and tuning it based on this feedback. This has become particularly important as superscalar, out-of-order processors have become prevalent which make it difficult to estimate or determine software bottlenecks.

The concerns raised about location transparency: memory access, partial failure, and concurrency are all addressed by using a fragmented object model where accesses are local by default, and any data serialization or failure handling is encapsulated by the object. Clients make local invocations and in the event that a request can be handled by a local

Table 4.1: Computation Latencies

Operation	Latency (ns)
Load from L1	1 [6]
Function Call, Load from L1, Return	3 [6]
Soft Page Fault (Linux)	500 [9]
RDMA Remote Read (Infiniband)	1500 [13]
TCP Round Trip over 10GbE	11000 (IX [22]), 20000 (EbbRT Paravirtualized), 30000 (Linux Paravirtualized)

representative, then no additional overheads are incurred. If a request cannot be handled locally, then it is the responsibility of the Ebb developers to ensure that, in aggregate, its representatives can satisfy the interface provided to the client as if it was local.

The most apparent limitation of the Elastic Building Block implementation is a language issue. One would like to define an abstract interface to an Ebb separately from any particular implementation. Then, clients can use the abstract interface and easily be constructed with a different implementation. In C++, the only way to define abstract interfaces on objects is through *pure virtual* classes. Doing so requires *dynamic dispatch*, every invocation looks up the corresponding function pointer in the virtual function table and invokes it. This is useful when, at runtime, a client wants to use many different implementations. However, this comes at a performance cost on each dispatch, and often times the flexibility is unnecessary. This is why we use C++ templates to enable static specialization. The code is written to support multiple implementations but a single implementation is chosen at compile time. The limitation, however is that C++ has no mechanism for defining an interface for a templated type. Best practices are to provide documentation on the required methods and members of a templated type. Failure to meet the requirements of the documentation may be caught at compile-time, if a developer is fortunate. Consumers do not specify what interface a templated class must provide, but rather the interface is defined by the manner in which the templated class is used. Other programming languages such as Rust and Haskell take a different approach. Their use of *traits* and *type classes* respectively require consumers of generic types to completely specify the interface they require. This allows the compiler to check that an implementation matches an interface at the invocation site and also forces developers to be explicit about the interfaces they expect. The C++ standardization committee has been considering various proposals to add *concepts*, an analogous language feature to C++ though no such proposal has been accepted yet.

Another issue with Ebbs is more systemic. A desirable property of object oriented programming is for everything to be an object. Otherwise, programmers are exposed to the difference between objects and other, non-object types. This is particularly the case in EbbRT because C++ already has a mechanism for extending a program through creating new C++ classes. Everytime a programmer wishes to create a new software component they must decide whether to make it an Ebb or not. In some cases, such as with a distributed service, the decision to make the component an Ebb is fairly obvious. In other cases, the decision is not so clear. Implicit to the definition of an Ebb is that references (`EbbRef`s to be specific) may be shared across multiple cores and machines. For some objects this may not make sense. Consider some object which provides an interface to an in-memory buffer. To support this interface across multiple cores, this object would have to synchronize on each access. To support the interface across multiple machines (with potential fault tolerance concerns), would require replication and/or partitioning of the data. While one could make all remote requests fail at runtime, the most obvious solution is not to use an Ebb. One can then use references which cannot be copied, such as C++'s `unique_ptr` and ensure that concurrent access is disallowed. Nothing prevents developers from building software which is not an Ebb, but these developers will also want to define interfaces which enable multiple implementations and flexible application composition. The root of this problem is that the Ebb primitive has two core responsibilities. First, it provides a mechanism to define a software component's interface. Second, it provides a mechanism to access per-core data and distribute functionality. Sometimes developers want the first property without the second. An encouraging avenue for further research is to explore further decomposing the responsibility of the Ebb primitive into multiple, orthogonal primitives.

The final limitation with the Ebb model is one common to many modular systems. How does a developer choose which implementations to use, which versions, and resolve all dependencies? Under our current implementation, this is a manual process. Ebbs are built as relocatable object files. To build a final binary, all dependencies must be resolved by linking all necessary Ebbs together in the final link step. A developer must explicitly specify dependent Ebbs (and their recursive dependencies) at link-time with (for example) a Makefile or some equivalent build tool. As Ebb applications grow, this may soon become unwieldy to maintain.

Note that this is no different from the typical manner in which C or C++ applications are constructed. We feel that this issue of managing many dependencies occurs in many C and C++ applications. This could be improved on with tooling which provides a central library package repository as is done in many languages like Javascript's `npm`, Haskell's `cabal` and Rust's `cargo`. All of these tools allow software libraries to explicitly specify their dependencies and automatically resolve recursive dependencies of an application (as well as fetching source or binaries from the internet).

In summary, Ebbs provide little more than a mechanism for information hiding which we extend to cover both local and distributed component implementations. Our design allows for low-overhead composition of even small Ebbs and yet provides flexibility for a wide-range of use-cases. While there are disadvantages, we feel that many of them can be addressed with further tooling and programming language support.

4.2 Primitives for Interface Construction

While the mechanism for invoking an Ebb must be efficient, it is equally important that Ebb interfaces permit both flexible and efficient implementations. In order to aide in the definition of Ebb interfaces, we provide several primitives which satisfy a variety of use

```
// Route and Send an Ethernet frame
Future<void> EthArpSend(uint16_t proto, const Ipv4Header& ip_header,
                        MutableIOBuf buf) {
  Ipv4Address local_dest = Route(ip_header.dst);
  Future<EthAddr> future_macaddr = ArpFind(local_dest); /* asynchronous call */
  return future_macaddr.Then(
    // continuation is passed in as an argument
    [buf = move(buf), proto](Future<EthAddr> f) { /* lambda definition */
      auto& eth_header = buf->Get<EthernetHeader>();
      eth_header.dst = f.Get();
      eth_header.src = Address();
      eth_header.type = htons(proto)
      Send(move(buf));
  });
}
```

Figure 4.3: Network code path to route and send an Ethernet frame.

cases yet do not inhibit performance. In this section we discuss the use of C++ lambdas and EbbRT Futures for structuring potentially asynchronous interfaces. We also discuss `IOBufs` which permit the transfer of data in a zero-copy fashion.

4.2.1 Lambdas and Futures

One of the core objectives of EbbRT's design is reducing development effort. Critics of event-driven programming point out several properties which place increased burden on the developer. One concern is that event-driven programming tends to obfuscate the control flow of the application [80]. For example, a call path that requires the completion of an asynchronous event will often pass along a callback function to be invoked when the event completes. The callback is invoked within a context different than that of the original call path, so it falls on the programmer to construct *continuations* i.e. control mechanisms used to save and restore state across invocations. C++ has recently added support for anonymous inline functions called *lambdas*. Lambdas can capture local state that can be referenced when the lambda is invoked. This removes the burden of manually saving and restoring state, and makes code easier to follow. We use lambdas in EbbRT to alleviate the burden of constructing continuations.

Another concern with event-driven programming is that error handling is much more complicated. The predominant mechanism for error handling in C++ is exceptions. When an error is encountered, an exception is thrown and the stack unwound to the most recent try/catch block which will handle the error. The automatic stack unwinding skips intermediate code which may not know how to handle the error. Because event-driven programming splits one logical flow of control across multiple stacks, exceptions must be handled at every event boundary. This puts a burden on the developer to catch exceptions at additional points in the code and either handle them or forward them to an error handling callback.

Our solution to this problem is our implementation of *Futures*. Figure 4.3 illustrates a code path in the EbbRT network stack. Line 5 issues a lookup into the ARP cache to translate an IP address to the corresponding MAC address. This may require an asynchronous ARP request to complete the translation. The `ArpFind` function returns a `Future<EthAddr>`. A future cannot be directly operated on. Instead, a function can be applied to it using the `Then` method (line 6). This function is invoked once the value is produced. The function receives a fulfilled future as a parameter and can use the `Get` method (line 10) to retrieve the underlying value. In the event that the MAC address translation is cached, this function is invoked synchronously.

The `Then` method of a future returns a new future representing the value to be returned by the applied function, hence the term monadic. This allows other software components to chain further functions to be invoked on completion. In this example, the `EthArpSend` method returns a `Future<void>` which merely represents the completion of some action, and provides no data.

Futures also aid in error processing. Each time `Get` is invoked, the future may throw an exception representing a failure to produce the value. If not explicitly handled, the future returned by `Then` will hold this exception instead of a value. The only invocation of `Then` that must handle the error is the final one, any intermediate exceptions will naturally flow to the first function which attempts to catch the exception. This behavior mirrors the behavior of exceptions in synchronous code. In this example, any error in ARP resolution will be propagated to the future returned by `EthArpSend` and handled by higher-level code.

Futures are used pervasively in interface definitions for Ebbs we have developed and lambdas are used in place of more manual continuation construction. Our experience using lambdas and futures has been positive. Initially, some members of our group had reservations about using these unfamiliar primitives as they hide a fair amount of potentially performance sensitive behavior. As we have gained more experience with these primitives, it has been clear that the behavior they encapsulate is common to many cases. Futures in particular encapsulate sometimes subtle synchronization code around installing a callback and providing a value (potentially concurrently). While this code has not been without bugs, we have more confidence in its correctness based on its use across EbbRT.

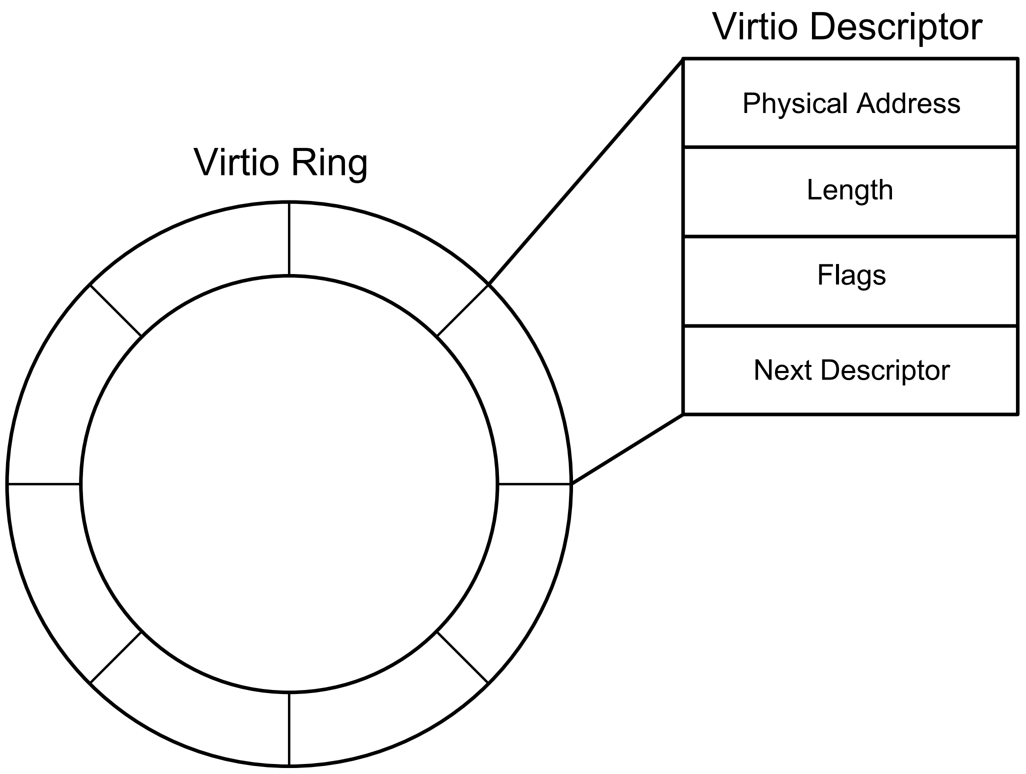
Futures are by no means a new programming language primitive. They originated as a primitive for use in the programming of distributed systems where the pipelining of futures (e.g. chaining of `Then` methods) could be used to hide-latency of remote operations [61]. Futures have become much more popular with the recent interest in asynchronous programming in domains such as web development and user interfaces. C++ has an implementation of futures in the standard library. Unlike our implementation, it provides no `Then` function, necessary for chaining callbacks. Instead users are expected to block on a future (using `Get`). Other languages such as C# and Javascript do provide monadic futures similar to ours and there are some implementations for C++ outside of the standard library.

Given the prevalence of event-driven programming in EbbRT, Futures take on a substantial role. They allow us to construct well-defined interfaces using an event-driven style while avoiding much of the complexity associated with event-driven programming.

4.2.2 IOBufs

Another common interface construction primitive is one to manage in-memory buffers. To motivate this primitive, consider the problem of mapping application-level I/O interfaces to hardware device semantics. As a motivating example, consider `memcached`, an in memory key-value store accessible via network requests. A `SET` message contains a header (common to all memcached messages), a key, and a value to be stored. A future `GET` operation with the corresponding key, should elicit a reply with the value.

Due to the single address space design, it should be possible for the data to flow from the network card, through the network stack, and up to the application without copying. It should also be possible for this zero-copy flow to be used in reverse, from application down to the network card. In fact, there is no need for the CPU to ever actually load the value portion of a memcached `SET` operation.

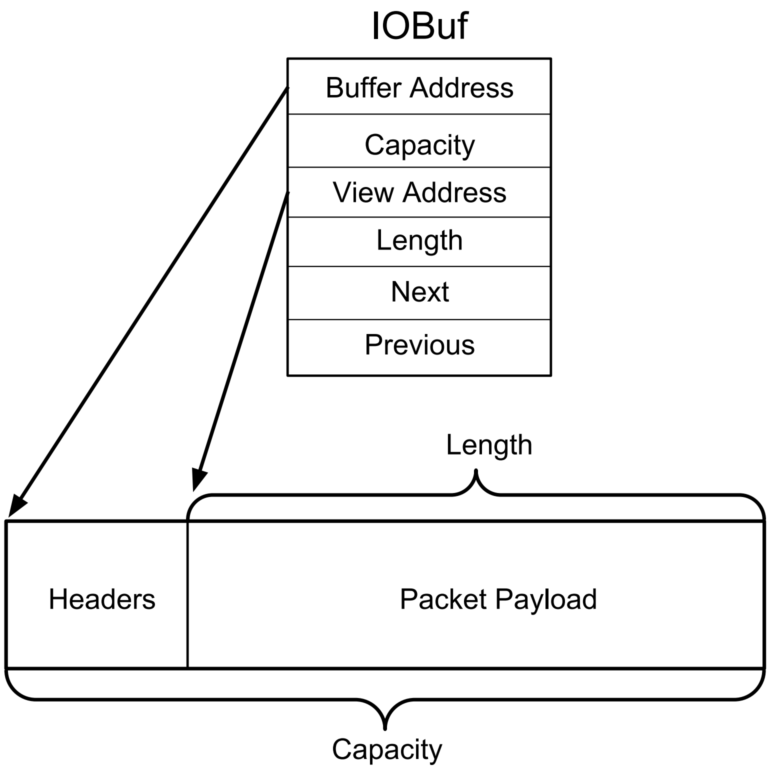


Most hardware network devices (and storage devices) interact with software through a ring buffer of descriptors each of which reference a contiguous region of physical memory. Descriptors can also chain together so that, for example, a single packet can be assembled from two distinct contiguous ranges of memory (for example, a header and a payload). Most modern hardware supports this ability (called scatter/gather) to construct a single logical element of data (e.g. a packet) from multiple memory regions and vice versa (splitting a packet across multiple memory regions). As an example, Figure 4.2.2 depicts the Virtio ring-buffer format. Virtio is a standard for para-virtualized (software implemented) devices, though it's interface is similar to a hardware network card. The guest operating system and the hypervisor communicate through the Virtio Ring, a ring-buffer of descriptors, each of which reference an extent of memory and potentially an additional descriptor (in order to create a chain).

To support developers, we created the `IOBuf` primitive, illustrated in figure 4.2.2. An `IOBuf` is a descriptor which manages ownership of a region of memory as well as a *view* of a portion of that memory. `IOBufs` can be chained to support scatter/gather interfaces. For example, the NIC driver creates an `IOBuf` and its associated memory and instructs the NIC to DMA a packet into the region of memory. When the DMA completes, the driver passes the `IOBuf` to the network stack. As the network stack processes the packet, it advances the view past the various headers within the packet and finally passes the `IOBuf` up to the application with the view containing only the payload.

When memcached receives a `SET` message, it advances the view to only contain the value portion of the message. Then, the `IOBuf` is stored in a table associated with the key. Later, when a `GET` operation occurs, memcached chains the value `IOBuf` to a new `IOBuf` containing a message header. This is sent to the network stack which then chains its own headers and finally the network card sends the entire chain out as one packet without copying.

The initial implemention of our `IOBufs` used reference counting. In this way they were very similar to existing buffer primitives like Linux's `skbuff` and BSD's `mbuf`. However, we found a number of cases where reference counting was unnecessary. For example, a memcached `GET` message contains a key to look up in order to construct a reply. The message can be destroyed as soon as the look up has completed. The `IOBuf` can be passed



from the network driver, through the network stack up to the application synchronously on a single core. However, to destroy the `IOBuf` associated with the `GET` request required an unnecessary atomic reference decrement. In order to avoid this, we modified `IOBufs` to only contain logic for view management and chaining. Memory management is separately implemented by a number of deriving classes (e.g. `UniqueIOBuf`, `SharedIOBuf`, `StaticIOBuf`).

Most consumers of `IOBufs` (e.g. the network stack) only care that as long as they hold an `IOBuf`, the memory remains valid regardless of exactly how the memory is managed. Therefore most interfaces are defined to take the base `IOBuf` class and memory management is hidden behind a virtual destructor. This allows consumers like the network stack to be written to accept any `IOBuf` and the application chooses how the memory of each network payload is managed. An alternative approach is for a buffer consumer (like the network stack) to explicitly notify a client that it is finished with a buffer via a callback. This is undesirable because it often requires additional resource tracking by both the consumer and the client.

`IOBufs` are a common element of Ebb interfaces. Any Ebb that produces or consumes large amounts of data such as those associated with networking, messaging, file access, and the `GlobalIdMap` use `IOBufs` to enable pervasive zero-copy I/O. We have also added additional support for ensuring serial reads and writes to `IOBufs`. For example, using a provided interface (called a `DataPointer`), developers can read or write a chained `IOBuf` without concerning themselves with handling data split across multiple chained buffers.

Memcached on Linux issues a separate `read` call for the message header, key and value which allows the application to separately manage the memory of each. This interface forces a data copy — which for Linux is already necessary due to the address space separation. However in EbbRT we want to encourage the application to continue zero-copy processing of the data which requires an expressive and ergonomic primitive. Ensuring that all use cases can be implemented efficiently supports the use of `IOBufs` throughout the system. The breadth of situations that they are used in increases reuse and composability of Ebbs.

4.2.3 Conclusion

Table 4.2 presents core system Ebbs provided by EbbRT as well as details about their implementation such as their use of the primitives described throughout this chapter as well as their representative structure. Ebbs enable developers to construct a wide-range of high-performance objects with implementation details hidden behind an interface. Our pervasive use of primitives such as Futures, Lambdas, and IOBufs simplify the definition of these interfaces and increases the interoperability of our Ebbs. Together, Ebbs and the primitives we have thus far developed enable EbbRT to be extended in a fine-grained fashion. Developers are free to modify the implementations of our existing Ebbs or define their own. Our primitives encourage succint and performant interfaces when doing so.

		Primitives			External Libraries				
		Futures	Lambdas	IOBufs	std c++	Boost	Intel TBB	capnproto	**Description**
Memory	PageAllocator				✓	✓	✓		Power of two physical page frame allocator
	VMemAllocator				✓				Allocates virtual address space
	SlabAllocator				✓	✓			Allocates fixed sized objects
	GeneralPurposeAllocator				✓				General purpose memory allocator
Objects	EbbAllocator				✓	✓			Allocates EbbIds
	LocalIdMap				✓	✓	✓		Local data store for Ebb data and fault resolution
	GlobalIdMap	✓		✓	✓			✓	Application-wide data store for Ebb data
Event	EventManager	✓	✓		✓	✓			Creates events and manages hardware interrupts
	Timer				✓	✓			Delay based scheduling of events
I/O	NetworkManager	✓	✓	✓	✓	✓			Implements TCP/IP stack
	SharedPoolAllocator				✓	✓			Allocates network ports
	NodeAllocator	✓			✓	✓		✓	Allocates, configures, and releases machines
	Messenger	✓	✓		✓				Cross node Ebb to Ebb communication
	VirtioNet			✓	✓				VirtIO network device driver

Table 4.2: The core Ebbs that comprise EbbRT. A gray row indicates that the Ebb has a multi-core implementation (one representative per core) while the others use a single shared representative.

Chapter 5

Compatibility

In the previous chapter we discussed how EbbRT encourages modularity in order to reduce development effort. In this chapter we discuss how we address compatibility with existing software and software tooling in order to further reduce application development burden. Every application has dependencies on software libraries or language runtimes which provide critical functionality. For example, some applications may make explicit system calls to invoke functionality provided by the Linux kernel. Others may target software libraries like the C or C++ standard libraries. Further applications may target managed runtimes like Java applications which must be executed by a Java Virtual Machine.

If EbbRT can provide compatibility with a broad set of software interfaces and development tools while still enabling high performance, we can further reduce developer effort. By providing compatibility with existing interfaces, developers can more easily port existing applications and software libraries. While much of EbbRT's value is derived from the ability to customize the system to a particular application for the sake of performance, we recognize that many aspects of an application are not performance-critical. Furthermore, rewriting an existing application to target novel interfaces is a substantial undertaking, particularly given the pace of modern application development where requirements and tradeoffs often change rapidly.

Our objective in supporting compatibility is to enable existing applications to run (potentially inefficiently) using EbbRT with little effort. Once a developer can integrate their application with EbbRT, they can selectively optimize their application. In particular, compatibility enables developers to take advantage of the modularity (to replace Ebbs as necessary) and efficient execution environment (to target more efficient abstractions) provided by EbbRT.

Software Application Programming Interfaces (APIs) provide the core functionality applications are developed on top of. These can range from low-level operating systems APIs like POSIX, to higher-level language runtimes like node.js or the Java Virtual Machine. We recognize that modern datacenter applications are written to wide variety of interfaces, in many different programming languages with distinct language runtimes. We recognize that these dependencies have a recursive property — A JVM may be implemented using the C standard library which is in turn may be implemented using Linux system calls. One option to provide compatibility for such a varied set of dependencies is to provide compatibility with some low-level common interface, such as the Linux userspace binary interface, and rely on the fact that most other APIs and language runtimes target such an interface. Indeed, this is the approach taken by OS^v [50] — a library operating system targeting cloud workloads. In particular, OS^v can link most Linux binaries unmodified. An outcome of this general approach of providing compatibility at a low-level is that any individual application relies on some subset of the functionality provided by a low-level API, but in aggregate, multiple applications rely on quite a broad set of functionality. For example, Tsai et al. [78] performed a study of the importance of Linux system calls. They analyzed Ubuntu 15.04 system installations on the Linux 3.20 kernel. This version of the Linux kernel has 320 system calls. The minimum system calls necessary to run a binary in the Ubuntu software repository is 40. To support half of the applications in the repository (weighted by their popularity), one needs to support 145 system calls. Our concern is that as functionality grows, the structure of the system ossifies. Even if an application doesn't use `fork`, for example, the fact that a system supports it, means the system has to track memory mappings, create address space isolation, reference count file descriptors, etc. And this in turn makes it more challenging to customize an application (simply because there is more unnecessary functionality). With EbbRT, we want to avoid this tension between customizability and compatibility.

Instead, we focus on enabling compatibility with a particular interface on a per-instance basis. For example, some EbbRT library operating systems may provide compatibility with the Java runtime, without needing to support the complete Linux system call API. Other instances may provide compatibility with some subset of POSIX. In doing so, we avoid the pitfalls of trying to build a system compatible with all interfaces simultaneously and yet preserve the ability to customize EbbRT library operating systems.

In the rest of this chapter, we describe two methods we use to provide compatibility without inhibiting a developer's ability to customize for high-performance:

1. First we adopt a heterogeneous distributed architecture which allows us to offload functionality to and from existing general purpose operating systems while keeping EbbRT library operating systems lightweight and easy to customize.

2. Second, we provide a compilation model and toolchain which provides standard interfaces (e.g. C and C++ standard libraries, sockets) in a manner that does not hinder the ability for developers to specialize library operating systems to their application.

5.1 Distributed Architecture

An EbbRT application is typically comprised of two binaries, a Linux binary linked to our provided library (the *hosted* environment) and a multiboot [8] compatible binary built using our modified GNU toolchain (the *native* environment). A user will then run the Linux binary which is responsible for launching native instances.

Software running on the native environment can communicate with the hosted environment in order to offload functionality. We encapsulate such behavior using Ebbs. For example, when software on the native environment invokes the `Filesystem` Ebb, a representative is faulted in which behaves by simply forwarding all requests to the hosted environment via the `Messenger` Ebb. The `Filesystem` Ebb representative on the hosted environment handles these messages and makes filesystem calls locally on the Linux system and sends the responses back to the native representative. In this way, we can provide filesystem interfaces to code running on the native environment with significantly reduced effort as compared to a complete implementation.

EbbRT is designed around the assumption that the hosted process and native library OSs can communicate via a logically isolated layer two (Ethernet) network. This assumption allows EbbRT library operating systems to be built to handle only trusted network traffic and not concern itself with denial of service attacks, malicious inputs, etc. We note that this isolation is fairly typical of existing datacenter technology. Public clouds like Amazon's EC2 and Google's Compute Engine provide isolated layer two networks using virtualization. Recent research projects like HIL [43] have demonstrated how this can be achieved without virtualization, instead providing isolation at the network switch through VLAN tagging.

The mechanism for allocating and launching new native library operating systems is provided by the `NodeAllocator` Ebb. The `NodeAllocator` has an `AllocateNode` method which takes as a parameter, the path to the bootable binary to be launched. The `AllocateNode` method returns a `NodeDescriptor` which includes a network identifier (typically an IP address) which can be used to send messages to via the `Messenger` Ebb.

The interface of the `NodeAllocator` Ebb is designed to allow EbbRT to be deployed on a number of different platforms such as commodity clouds (e.g. Google Compute Engine [4] or Amazon Web Services [1]) as well as hardware provisioning services such as Kittyhawk [17] or HIL [43]. In particular, the key method is `AllocateNode` which takes a path to the library OS binary as well as a hardware configuration (number of CPUs, amount of RAM, etc.) and launches the given binary on the isolated network. A `NodeDescriptor` is provided as a return value which contains a `NetworkId` suitable for the application to send a message to. Whether these library operating systems are launched on physical hardware or virtual machines is an implementation detail.

Our default `NodeAllocator` implementation uses the Docker [5] containerization platform to launch EbbRT library operating systems. A Docker container provides an isolated process, network and file space for an application. Multiple containers can run on the same kernel, merely providing an isolated userspace. When the `NodeAllocator` is initialized it allocates a new layer two network for the process and it's library operating systems. In the default deployment of Docker, these networks are Linux bridges which are NATed to the external network of the host server. However, with Docker Swarm (a distributed version of Docker), these are overlay networks which can span multiple hosts. The current implementation of these overlay networks sends layer two data between hosts via UDP. Docker also has has experimental support for VLANs. The rest of EbbRT is not tied to this particular implementation of the `NodeAllocator`.

5.1.1 Advantages

Function offloading using a distributed architecture is not a new technique. Two systems which motivated our approach are the Bluegene/P supercomputer [30] and the Libra library operating system [15]. The Bluegene/P supercomputer runs a Compute Node Kernel which is a lightweight kernel, highly optimized for executing supercomputing workloads and in particular, applications written to the Message Passing Interface (MPI). For example, the compute node kernel does not handle any interrupts and simply maps the network device buffers into userspace. Non-performance critical operations like reading a file are offloaded to other machines running a modified Linux kernel called the I/O node kernel. Libra is a lightweight library operating system for the Java Virtual Machine. An instance of Libra executes in a virtual machine on the Xen [20] virtual machine monitor and offloads functionality to a Linux instance executing in a virtual machine on the same host. The Linux instance provides networking and file access which allows Libra to be more optimized for the JVM. The main difference between these approaches, and EbbRT's is that both Libra and Bluegene target a fixed interface for compatibility (the JVM and a subset of POSIX, respectively). On the other hand, our objective in EbbRT is to allow developers to provide compatibility as required by their application. The first advantage of our approach is that our method for providing compatibility is part of the EbbRT programming model. Rather than provide compatibility with any fixed interface, we provide a general method for doing so and allow developers to extend functionality as necessary.

The second advantage of this distributed architecture is that it allows us to offload functionality from EbbRT's native execution environment to an existing operating system for compatibility. For example, we constructed the `Filesystem` Ebb which provides standard POSIX filesystem operations (e.g. `open`, `stat`, `read`, `write`). The default implementation of this Ebb creates native representatives which simply forward all requests to the hosted implementation using the `Messenger` Ebb. The hosted representative issues the corresponding system call which is served by the Linux kernel. The response is then sent back (via the `Messenger` to the native representative which fulfills the original invocation. This is useful, not as an interface for local disk access (which we would rather provide lower-level interfaces for) — but for compatibility with non-performance critical filesystem access such as reading of configuration or logging messages.

This code is simple to write and allows us to rapidly develop applications which depend on filesystem access. A functionally complete POSIX filesystem requires more than just data management, but also users, groups, and permissions. By offloading the functionality through the hosted environment, file access occurs with the authority of the user executing the hosted process. Furthermore, an application which does not require file access, can simply not include the `Filesystem` Ebb.

This leads to the third and final advantage of this architecture which is that we are able to reuse Ebbs to address compatibility. We use Ebbs to encourage developers to customize their application for performance. However in this scenario, we use Ebbs to extend the functionality of EbbRT for compatibility. The fact that Ebbs provide a shared mechanism to achieve both compatibility and customizability is what allows us to incrementally optimize applications. We can provide a trivial implementation of an Ebb for the sake of compatibility and then incrementally optimize the implementation of that Ebb for a particular application or use-case. Because of the encapsulation provided by an Ebb interface, this optimization can be done independently from the rest of the application.

Conversely, one can view our hybrid distributed architecture as providing a manner in which to incrementally accelerate existing applications. An application can be accelerated by linking with the hosted library and launching specialized library operating systems to perform some function of the application. This is analagous to the use of GPUs to accelerate heavily data-parallel portions of an application. Similarly, in both instances developers must balance the computation speedup with the communication overheads.

To demonstrate the advantages of this approach, we extended Sage [10]. Sage is an open source mathematics environment similar to Matlab. It provides many common math library routines and objects through a Python interface (typically accessed via an interactive shell). We used EbbRT to transparently distribute and accelerate particular matrix operations within Sage.

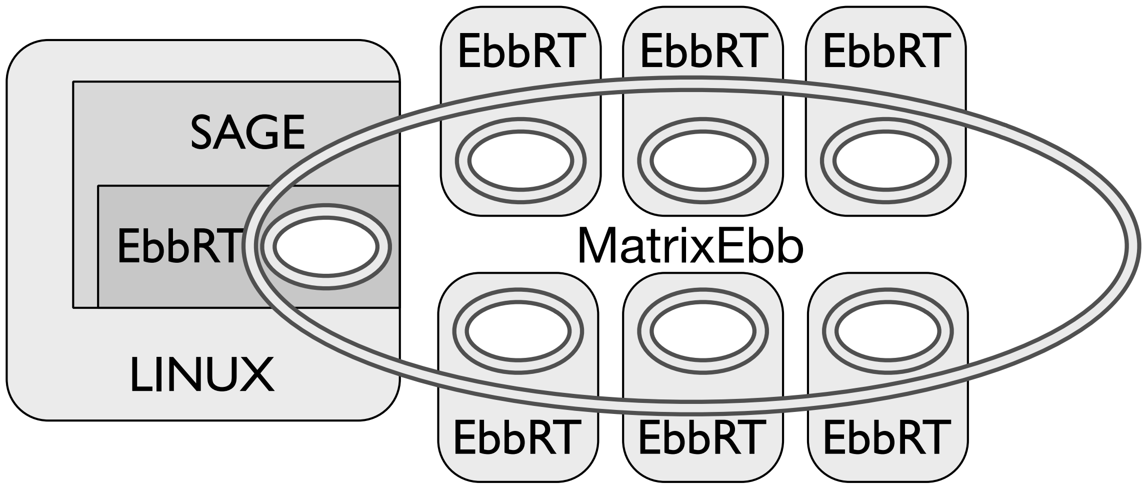


Figure 5.1: EbbRT Sage Matrix Integration.

We created a Python module which can be dynamically loaded into the Sage environment. This module links with the EbbRT Linux library and provides a python matrix object which simply wraps a matrix Ebb. When this python matrix is instantiated at the command line, an instance of the matrix Ebb is constructed to back it. When calls are made to the python matrix object, they are forwarded to the matrix Ebb which may internally distributed its functionality to satisfy its interface. Figure 5.1 illustrates the realized runtime structure.

In our particular matrix Ebb implementation, the representative running within the

Sage process on Linux allocates nodes from the `NodeAllocator` booted with an EbbRT library OS designed to hold a fixed tile of the matrix values and perform the core computations on that matrix tile. The matrix Ebb links with the Boost uBLAS library to provide local matrix operations. Nodes are allocated lazily, when an operation requires a particular portion of the matrix for the first time. This structure allows for matrix operations to be done both lazily and in parallel. For example, as matrix elements are set, the Linux representative will allocate nodes as necessary to store the tile of the matrix that the element belongs to. Operations of the matrix Ebb such as element-wise randomization can naturally be done in parallel across the tiles. Our matrix Ebb implements a number of matrix operations such as summation, multiplication, element-wise randomization, and element access.

An EbbRT library OS is well suited for offloading computationally expensive functionality because it allows the application complete control of the hardware. For example, interrupts are disabled which prevents context switches from causing cache pollution to slow down the computation. Additionally, complete control over memory allows the use of large pages to reduce TLB contention. Such optimizations are similar to those explored in the construction of operating systems for high performance computing like CNK [69] and FusedOS [71].

From the perspective of a user at the Sage console, the matrix behaves just as any other python object. In fact, if an instance of the matrix object is garbage collected (perhaps due to the python variable going out of scope), the underlying Ebb is destroyed and any nodes that were allocated are freed to the NodeAllocator. This is a feature of the particular matrix ebb implementation. A different implementation may colocate matrices on the same nodes in which case its destruction logic would encapsulate the dependency. Ebb encapsulation ensures that such differences in implementation would not impact Sage or the python module.

In summary, the EbbRT distributed architecture allows developers to extend the functionality of a library operating system for compatibility or, vice-versa, to accelerate the functionality of existing applications. This is all encapsulated behind Ebbs which allows for incremental development.

5.1.2 Disadvantages

One disadvantage of our distributed architecture is that our deployment model assumes the existence of some system capable of allocating isolated networks and (virtual) machines. At the outset of this project, this deployment model was limited to a small number of virtualized cloud environments like Amazon Web Services (AWS) or vCloud [51]. When we initially started development on this model, we had to develop a service with these capabilities which we could run on our local development server. In the time since we started, this model of deployment has become substantially more popular with the development of container systems like Docker and Kubernetes. With the ubiquity of public and private clouds (using virtualization, containers, or a combination) — we feel EbbRT's distributed architecture applies to a broad set of datacenter environments.

The second major disadvantage of our approach is that not all compatibile interfaces can be easily offloaded. Consider UNIX signals which provide an asynchronous notification to notify a process that an event has occurred. For example, a program can request that the `SIGALARM` signal fires after a specified number of seconds elapses by making the `alarm` system call. EbbRT's events are non-preemptible so there is no way to interrupt an event to invoke a signal handler. It is not particularly challenging to develop preemptible threads on top of EbbRT's events which can then provide a compatible implementation of signals. However, functionality offload does not provide any help in implementing signals — the challenge is particular to the execution model of the EbbRT native execution environment, the functionality itself is fairly rudimentary.

The final disadvantage of our distributed architecture is that when offloading functionality, implementors must consider new challenges like scalability and an expanded fault domain. For example, consider the behavior of a file read be if the machine where the hosted environment executes crashes. It is not clear that EbbRT needs distribution to provide function offloading. For example, FusedOS [71] and mOS [82] both adopt an approach where the resources on a single machine can be partitioned across light-weight kernels and full-weight kernels. For example, one can run a fully functional Linux kernel across some set of cores of a machine and then run a custom, light-weight kernel on other cores. These can communicate via some form of message passing or shared memory in order to provide function offloading. Similarly, Dune [21] provides Linux system calls to hardware-virtualized environments where one can run custom operating systems. All of these approaches provide many of the same advantages of our distributed architecture, but the functionality offloading is performed to the local host.

5.2 Compilation model

In addition to providing compatibility by function offloading through our distributed architecture, EbbRT also directly provides compatibility with many existing libraries and build systems through our compilation model.

EbbRT is distributed as a software library (`libebbrt`) and a modified GNU toolchain. The software library can be built to target Linux and linked to existing Linux applications. The native toolchain builds a `sysroot` where EbbRT is installed as a library. The compiler is modified to implicitly link against `libebbrt` and use a custom linker script which ensures the address layout as EbbRT expects. Therefore, building new libraries or applications just requires compiling with the `x86_64-ebbrt` target of our modified compiler. In effect, libraries and applications are cross-compiled to EbbRT — but produce a bootable ELF linked with the runtime. Given that nearly all build systems provide some support for cross-compilation, integrating new libraries and applications with EbbRT often requires no build-system changes.

The toolchain provides a port of the `newlib` C standard library. The C standard library has a number of functions which require runtime support, for example `printf` requires some form of I/O. Newlib reduces this support to 22 functions (e.g. `open`, `read`, `malloc`, etc.)

which the target operating system provides. We define these symbols, but the default implementation of many of these methods will cause a system abort. Applications which need such functionality must override these methods as neccessary.

We also provide `libstdc++`, the GNU C++ standard library. All of the runtime dependencies are provided by the C standard library. However, C++ requires some runtime support for language features such as scoped static variable initialization (which requires locking to ensure only one thread initializes the variable at a time) and exceptions (which requires stack unwinding). We provide the necessary functionality. For flexibility, these can be overriden at link-time.

With the C and C++ toolchain we are able to adopt many common libraries in the construction of EbbRT library operating systems. For example, we make use of the Boost C++ libraries and the Intel Threading Building Blocks (TBB) library in several Ebbs as shown in Table 4.2.

5.2.1 Advantages

The decision to distribute a toolchain with a custom system target is not an obvious one. Most kernels provide their own build system which produces a single artifact (the kernel) and headers necessary to construct applications that utilize and interact with the kernel. Even extensible systems often create an assumption about the manner in which it must be extended. Consider the way a Linux kernel module is built. One must invoke the kernel's Makefile when building a module. We initially started down a similar path, each library or application we sought to use within EbbRT had to include our own Makefile. We found that this would not scale to large applications which already have their own complicated build systems.

Furthermore, most operating systems do not integrate with the C or C++ standard libraries. Rather, they require a limited subset of this functionality which is typically reimplemented within each operating system. Given our desire to integrate with existing software, we had to adopt the C and C++ standard libraries. We found that this greatly simplified the development of system functionality - for example the `SlabAllocator` Ebb uses Boost's intrusive data structures library.

The general advantage of our compilation model is that our approach closely mirrors the standard way of constructing C and C++ applications rather than the approach taken by many operating systems. This enables us to reuse existing software libraries and makes application developers much more comfortable developing on EbbRT. Our approach has the added advantage that it makes it simpler to develop system functionality.

5.3 Conclusion

One of the primary ways EbbRT reduces development effort is by providing compatibility with existing applications, software interfaces and build tooling. Our compatibility does not come at the expense of the ability to specialize the system for high performance. In particular, we use function offloading with our distributed architecture to cleanly isolate rich functionality necessary for compatibility from performance critical functionality. We also adopt a standard compilation model which brings application development practices to the construction of library operating systems. Together, these approaches enable developers to execute full-featured applications on top of the same base system which enables a high degree of optimization.

Chapter 6

Evaluation

One goal of EbbRT is to enable the construction of high-performance library operating systems. To evaluate this, we compare the performance of EbbRT library operating systems primarily with the Linux general purpose operating system. The other goal of EbbRT is to enable this degree of performance without inflicting a large burden on developers. We evaluate EbbRT based on indicators of development and maintenance effort. This primarily involves describing code complexity, component reuse, and opportunities for offloading functionality.

Through evaluating EbbRT we aim to affirm that our implementation fulfills the following three objectives: 1. supports high-performance specialization, 2. provides support for a broad set of applications, and 3. simplifies the development of application-specific systems software.

We run our evaluations on a cluster of servers connected via a 10GbE network and commodity switch. Each machine contains two 6-core Xeon E5-2630L processors (run at 2.4 GHz), 120 GB of RAM, and an Intel X520 network card (82599 chipset). The machines have been configured to disable Turbo Boost, hyper-threads, and dynamic frequency scaling. Additionally, we disable IRQ balancing and explicitly assign NIC IRQ affinity. For the evaluation, we pin each application thread to a dedicated physical core.

Each machine boots Ubuntu 14.04 (trusty) with Linux kernel version 3.13. The EbbRT native library OSs are run as virtual machines, which are deployed using QEMU (2.5.0) and the KVM kernel module. In addition, the VMs use a `virtio-net` paravirtualized network card with support of the `vhost` kernel module. We enable multiqueue receive flow steering for multicore experiments. Unless otherwise stated, all Linux applications are run within a similarly configured VM and on the same OS and kernel version as the host.

The performance evaluations are broken down as follows: 1. micro-benchmarks designed to quantify the base overheads of the primitives in our native environment and 2. macro-benchmarks that exercise EbbRT in the context of real applications. While the EbbRT hosted library is a primary component of our design, it is not intended for high-performance, but rather to facilitate the integration of functionality between a general purpose OS process and native instances of EbbRT. Therefore, we focus our performance evaluation on the EbbRT native environment.

6.0.1 Microbenchmarks

The first micro-benchmark evaluates the direct cost of Ebb invocation as compared to other function call overheads. The seconds micro-benchmark evaluates the memory allocator and aims to establish that the overheads of our Ebb mechanism do not preclude the construction of high-performance components. The third set of micro-benchmarks evaluate the latencies and throughput of our network stack and exercise several of the system features we've discussed, including idle event processing, lambdas, and the `IOBuf` mechanism.

6.0.1.1 Ebb Invocation

Figure 6.1 shows the overhead of Ebb dispatch as compared to standard C++ object dispatch. The microbenchmark measures 1000 invocations of an object with an empty function. The "Inline" row shows the cost of a C++ inlinable method invocation. The "No Inline" row shows the cost where inlining of the method is explicitly disallowed. The "Virtual" row shows the cost when the method is declared as `virtual` and compiler devirtualization is disabled. The final row shows the cost of an Ebb dereference and dispatch to an inlinable method.

These results demonstrate that Ebb usage does not significantly hinder performance. They can be used for a fine-grained decomposition without concern. The usage of a virtual

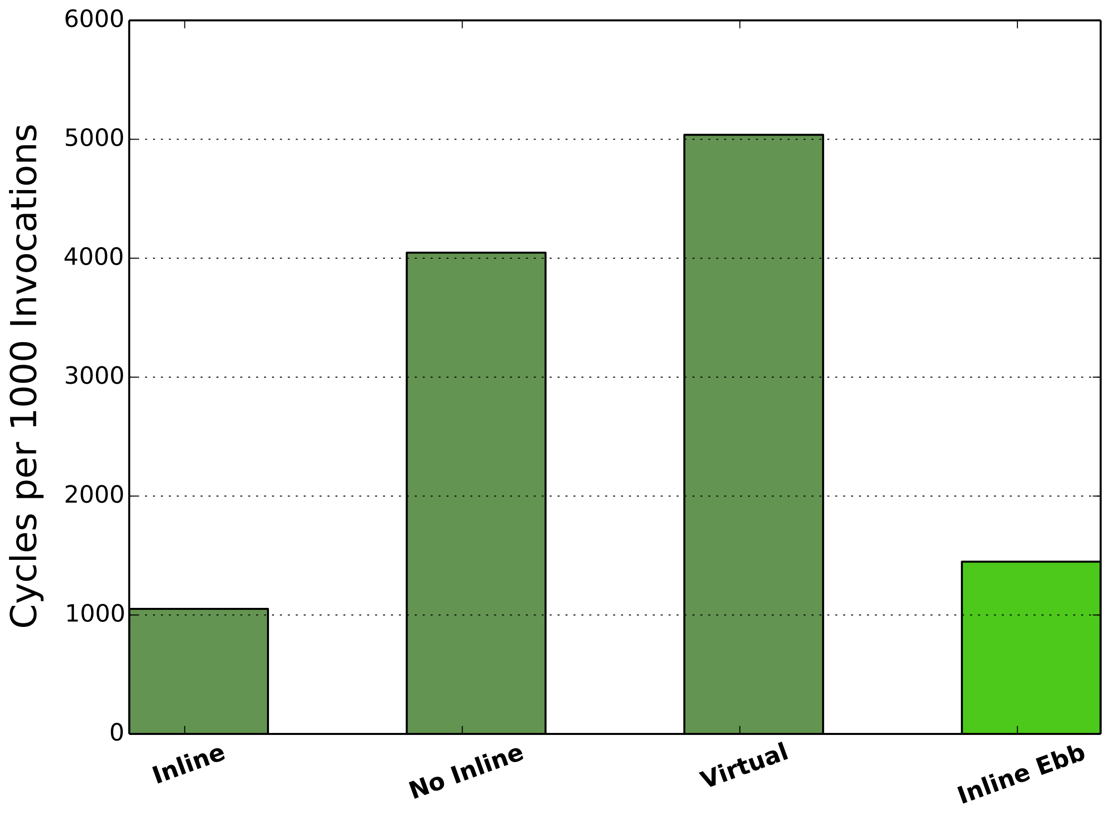


Figure 6.1: Object dispatch costs for 1000 invocations

memory region to enable lookup for per-core representatives as well as allowing inlining ensures the primitive is efficient.

We also measured the cost of invoking an Ebb within our hosted environment which cannot use a virtual memory region and instead must do a lookup into a per-core hash table. We found Ebb dereference and invocations under Linux to be roughly 19 times the cost. This is not a significant concern as the hosted environment is largely used for compatibility and not performance-critical software.

These results highlight an additional important benefit of the EbbRT approach to library OSs. Unlike OSs that provide an ABI and use runtime linking, EbbRT code paths get statically integrated and optimized along with the application code by the compiler; this allows the compiler to create higher quality end-to-end code paths. Unlike library

OSs that purely target binary compatibility, EbbRT enables developers to maximize the benefits of customization.

6.0.1.2 Memory Allocation

K42 [52] did not define its memory allocator as a fragmented object because the invocation overheads (e.g., virtual function dispatch) were thought to be too expensive. A goal for the design of our Ebb mechanism is to provide near-zero overhead so that all components of the system can be defined as Ebbs.

The costs of managing memory is critical to the overall performance of an application. Indeed, custom memory allocators have shown substantial improvements in application performance [24]. We ported *threadtest* from the Hoard [23] benchmark suite to EbbRT in order to compare the performance of the default EbbRT memory allocator to that of the glibc 2.2.5 and jemalloc 4.2.1 allocators.

In *threadtest*, each thread t allocates and frees $\frac{N}{t}$ 8 byte objects. This task is repeated for i iterations. Figure 6.2 shows the cycles required to complete the workload across varying amounts of threads. We run *threadtest* in two configurations. In configuration *I.*, the number of objects, N, is large, while the number of iterations is small. In configuration *II.* the number of objects is smaller and the iteration count is increased. The total number of memory operations is the same across both configurations.

In the figure we see EbbRT's memory allocator scales competitively with the production allocators. Our scalability advantage is in part due to locality enabled by the per-core Ebb representatives of the memory allocator and our lack of preemption which remove any synchronization requirements between representatives. The jemalloc allocator achieves similar scalability benefits by avoiding synchronization through the use of per-thread caches.

This comparison is not intended to establish the EbbRT memory allocator to be the best in all situations, nor is it an exhaustive memory allocator study. Rather, we aim to demonstrate that the overheads of the Ebb mechanism do not preclude us from the construction of high-performance components.

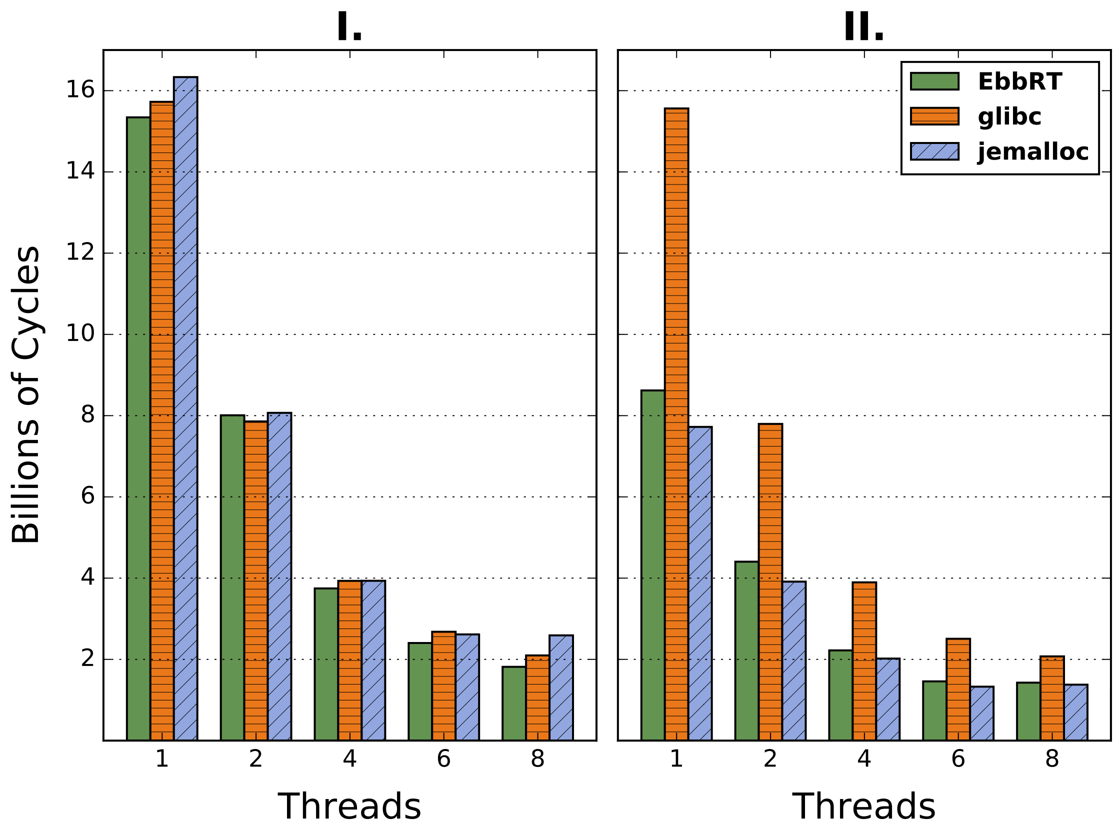


Figure 6.2: Hoard Threadtest. I.) $N = 100,000$, $i = 1000$; II.) $N = 100$, $i = 1000000$

The EbbRT memory allocator is defined as an Ebb using per-core representatives for locality. Due to the lack of pre-emption, the per-core data does not require synchronization. This result is not intended to establish that the EbbRT memory allocator is the best in all situations. Rather, we demonstrate that the general Ebb mechanisms allow us to define an Ebb interface to the memory allocator and the overhead of these mechanisms doesn't preclude the construction of high-performance components.

6.0.1.3 Network Stack

To evaluate the performance of our network stack we ported the NetPIPE [76] and iPerf [77] benchmarks to EbbRT. NetPIPE is a popular ping-pong benchmark where a client sends a fixed-size message to the server, which is then echoed back after being completely received.

In the iPerf benchmark, a client opens a TCP stream and sends fixed-size messages which the server receives and discards. With small message sizes, the NetPIPE benchmark illustrates the latency of sending and receiving data over TCP. The iPerf benchmark confirms that our run-to-completion network stack doesn't preclude high throughput applications. An EbbRT iPerf server was shown to saturate our 10GbE network with a stream of 1 kB message sizes.

Figure 6.3 shows NetPIPE goodput achieved as a function of message size. Two EbbRT servers achieve a one-way latency of 24.53 µs for 64 B message sizes and are able to attain 4 Gbps of goodput with messages as small as 100 kB. In contrast, two Linux VMs achieve a one-way latency of 34.27 µs for 64 B message sizes and required 200 kB sized messages to achieve equivalent goodput.

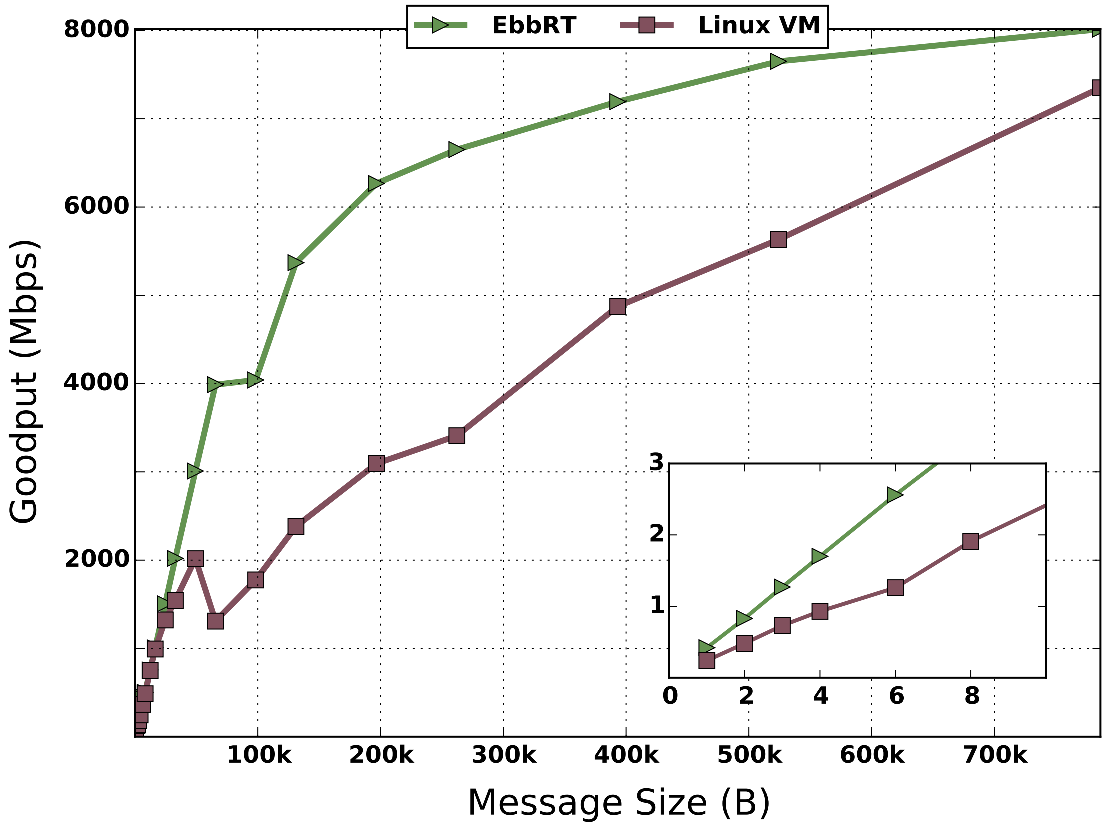


Figure 6.3: NetPIPE performance as a function of message size. Inset shows small message sizes.

With small messages, both systems suffer some additional latency due to hypervisor processing involved in implementing the paravirtualized NIC. However, EbbRT's short path from (virtual) hardware to application achieves a 40% improvement in latency with NetPIPE. This result illustrates the benefits of a non-preemptive event-driven execution model and zero-copy instruction path. With large messages, both systems must suffer a copy on packet reception due to the hypervisor, but EbbRT does no further copies, whereas Linux must copy to user-space and then again on transmission. This explains the difference in Netpipe goodput before the network becomes the bottleneck.

Across a series of iPerf runs, we observed on average a 1.8 increase in achievable throughput on EbbRT (9.49 Gbit/sec with 8kb messages) compared to a Linux VM (9.32 Gbit/sec with 8kb messages. The iPerf results demonstrate that the EbbRT network performance does not preclude the use of high-throughput applications.

EbbRT:	9.49	Linux VM:	9.32

Table 6.1: Throughput Gbits/sec (standard iPerf client, 8kb messages)

6.0.2 Memcached

We evaluate memcached[36], an in-memory key-value store that has become a common benchmark in the examination and optimization of networked systems. Previous work has shown that memcached incurs significant OS overhead[48], and hence is a natural target for OS customization. Rather than port the existing memcached and associated event-driven libraries to EbbRT we re-implemented memcached, writing it directly to the EbbRT interfaces.

Our memcached implementation is a multi-core application that supports the standard memcached binary protocol. In our implementation, TCP data is received synchronously from the network card and passed up to the application. The application parses the client request and constructs a reply, which is sent out synchronously. The entire execution path, up to the application and back again, is run without pre-emption. Key-value pairs

	Request/sec	Inst/cycle	Inst/request	LLC ref/cycle	I-cache miss/cycle
EbbRT	379387	0.81	5557	0.0081	0.0079
Linux VM	137194	0.71	13604	0.0098	0.0339

Table 6.2: Memcached CPU-efficiency metrics

are stored in an RCU hash table to alleviate lock contention, a common cause for poor scalability in memcached. Our implementation of memcached totals 361 lines of code. We lack some features of the standard memcached (namely authentication and some of per-key commands such as queue operations), but are otherwise protocol compatible. Functionality support has been added incrementally as needed by our workloads.

We compare our EbbRT implementation of memcached, run within a VM, to the standard implementation (v.1.4.22) run within a Linux VM, and as a Linux process run natively on our machine. We use the `mutilate`[56] benchmarking tool to place a particular load on the server and measure response latency. We configure `mutilate` to generate load representative of the Facebook ETC workload[19], which has 20 B–70 B keys and most values sized between 1 B–1024 B. All requests are issued as separate memcached requests (no `multiget`) over TCP. The client is configured to pipeline up to four requests per TCP connection. We dedicate 7 machines to act as load-generating clients for a total of 664 connections per server.

Figure 6.4 presents the 99th percentile latency as a function of throughput for single core memcached servers. At a 500 µs 99th percentile Service Level Agreement (SLA), single core EbbRT is able to attain 1.88× higher throughput than Linux within a VM. EbbRT outperforms Linux running natively by 1.15×, even with the hypervisor overheads incurred. Additionally, we evaluated the performance of OS^v[50], a general purpose library OS that similarly targets cloud applications run in a virtualized environment. OS^v differs from EbbRT by providing a Linux ABI compatible environment, rather than supporting a high-degree of specialization. We found that the performance of `memcached` on OS^v was not competitive with either Linux or EbbRT with a single core. Additionally, OS^v's performance degrades when scaled up to six cores (omitted from figure 6.5) due to a lack

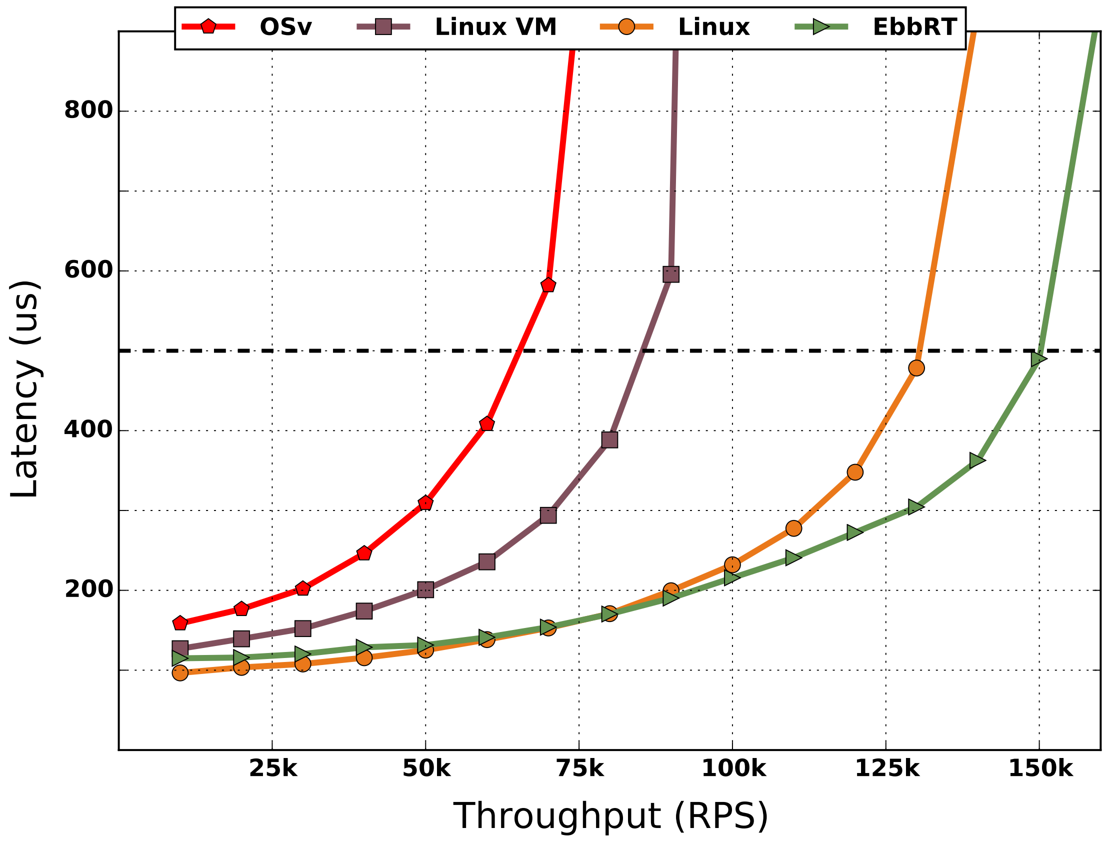


Figure 6.4: Memcached Single Core Performance

of multiqueue support in their `virtio-net` device driver.

Figure 6.5 presents the evaluation of memcached running across six cores. At a 500 µs 99th percentile SLA, six core EbbRT is able to attain a 2.08× higher throughput than Linux within a VM and 1.50× higher than Linux native. To eliminate the performance impact of application-level contention, we also evaluated memcached run natively as six separate processes, rather than a single multithreaded process ("Linux (process)" in Figure 6.5). EbbRT outperforms the multiprocess memcached by 1.30× at 500 µs 99th percentile SLA.

To gain insight into the source of EbbRT's performance advantages, we examine the CPU-efficiency of the memcached servers. We use the Linux Kernel `perf` utility to gather data across a 10 second duration of a fully-loaded single core memcached server run within a VM. Table 6.2 presents these statistics. We see that the EbbRT server is processing requests

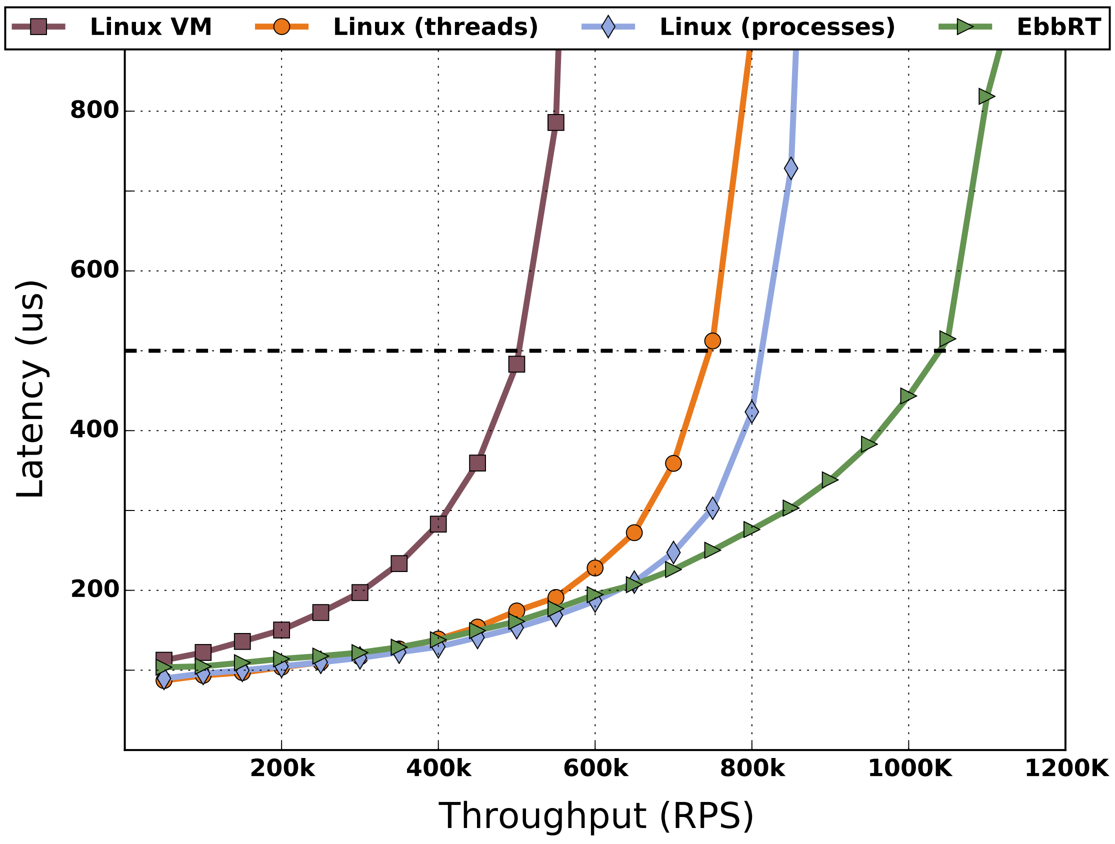


Figure 6.5: Memcached Multicore Performance

at 2.75× the rate of Linux. This can be largely attributed to our shorter non-preemptive instruction path for processing requests. Observe that the Linux rate of instructions per request is 2.44× that of EbbRT. The instructions per cycle rate in EbbRT, a 12.6% increase over Linux, shows that we are running more efficiently overall. This can be again observed through our decreased per-cycle rates of last level cache (LLC) reference and icache misses, which, on Linux, increase by 1.21× and 4.27×, respectively.

The above efficiency results suggest that our performance advantages are largely achieved through the construction of specialized system software to take advantage of properties of the memcached workload. We illustrate this in greater detail by examining the per-request latency for EbbRT and Linux (native) broken down into time spent processing network ingress, application logic, and network egress. For Linux, we used the `perf` tool to gather

	Ingress	Application	Egress	Total
EbbRT	0.89 µs	0.86 µs	0.83 µs	2.59 µs
Linux	1.05 µs	1.30 µs	1.46 µs	3.81 µs

Table 6.3: Memcached Per-Request Latency

	Inst/cycle	LLC ref/cycle	TLB miss/cycle	VM exit	Hypervisor time	Guest kernel time
EbbRT	2.48	0.0021	1.18e-5	5950	0.33%	N/A
Linux VM	2.39	0.0028	9.92e-5	66851	0.74%	1.08%

Table 6.4: V8 JavaScript Benchmark CPU-efficiency metrics

stacktrace samples over 30 seconds of a fully loaded, single core memcached instance and categorized each trace. For EbbRT, we instrumented the source code with timestamp counters. Table 6.3 presents this result. It should be noted that, for Linux, the "Application" category includes time spent scheduling, context switching, and handling event notification (e.g. `epoll`). The latency breakdown demonstrates that the performance advantage comes from specialization across the entire software stack, and not just one component.

By writing to our interfaces, memcached is implemented to directly handle memory filled by the device, and can likewise send replies without copying. A request is handled synchronously from the device driver without pre-emption, which enables a significant performance advantage. EbbRT primitives, such as `IOBufs` and RCU data structures, are used throughout the application to simplify the development of the zero-copy, lock-free code.

In the past, significant effort has gone into improving the performance of memcached and similar key-value stores. However, many of these optimizations require client modifications [59, 70] or the use of custom hardware [60, 46]. By writing memcached as an EbbRT application, we are able to achieve significant performance improvements while maintaining compatibility with standard clients, protocols, and hardware.

6.0.3 Node.js

It is often the case that specialized systems can demonstrate high performance for a particular workload, such as packet processing, but fail to provide similar benefits to more full-featured applications. A key objective of EbbRT is to provide an efficient base set of primitives on top of which a broad set of applications can be constructed.

We evaluate node.js, a popular JavaScript execution environment for server-side applications. In comparison to memcached, node.js uses many more features of an operating system, including virtual memory mapping, file I/O, periodic timers, etc. Node.js links with several C/C++ libraries to provide its event-driven environment. In particular, the two libraries which involved the most effort to port were V8[42], Google's JavaScript engine, and libuv[7], which abstracts OS functionality and callback based event-driven execution.

Porting V8 was relatively straightforward as EbbRT supports the C++ standard library, on which V8 depends. Additional OS functionality required such as clocks, timers, and virtual memory, are provided by the core Ebbs of the system. Porting libuv required significantly more effort, as there are over 100 functions of the libuv interface which require OS specific implementations. In the end, our approach enables the libuv callbacks to be invoked directly from a hardware interrupt, in the same way that our memcached implementation receives incoming requests.

The effort to port node.js was significantly simplified by exploiting EbbRT's model of function offloading. For example, the port included the construction of an application-specific `FileSystem` Ebb. Rather than implement a file system and hard disk driver within the EbbRT library OS, the Ebb calls are offloaded to a (hosted) representative running in a Linux process. Our default implementation of the `FileSystem` Ebb is naïve, sending messages and incurring round trip costs for every access, rather than caching data on local representatives. For evaluation purposes we use a modified version of the `FileSystem` Ebb which performs no communication and serves a single static node.js script as `stdin`. This implementation allows us to evaluate the following workloads (which perform no file access) without also involving a hosted library.

One key observation of the node.js port is the modest development effort required to get a large piece of software functional, and, more importantly, the ability to reuse many of the software mechanisms used in our memcached application. The port was largely completed by a single developer in two weeks. Concretely, node.js and its dependencies total over one million lines of code, the majority of which is the v8 JavaScript engine. We wrote about 3000 lines of new code in order to support node.js on EbbRT. A significant factor in simplifying the port is the fact that EbbRT is distributed with a custom toolchain. Rather than needing to modify the existing node.js build system, we specified EbbRT as a target and built it as we would any other cross compiled binary. This illustrates EbbRT's support for a broad class of software as well as the manner in which we reduce developer burden required to develop specialized systems.

6.0.3.1 V8 JavaScript Benchmark

To compare the performance of our port to that of Linux, we launch node.js running version 7 of the V8 JavaScript benchmark suite [41]. This collection of purely compute-bound benchmarks stresses the core performance of the V8 JavaScript engine. Figure 6.6 shows the benchmark scores. Scores are computed by inverting the running time of the benchmark and scaling it by the score of a reference implementation (higher is better). The overall score is the geometric mean of the 8 individual scores. The figure normalizes each score to the Linux result.

EbbRT outperforms Linux run within a VM on each benchmark, with a 5.1% improvement in overall score. Most prominently, EbbRT is able to attain a 30.3% improvement in the memory intensive `Splay` benchmark. As we've made no modification to the V8 software, just running it on EbbRT accounts for the improved performance.

We further investigate the sources of the performance advantage by running the Linux `perf` utility to measure several CPU efficiency metrics. Table 6.4 displays these results. Several interesting aspects of this table deserve highlighting. First, EbbRT has a slightly better IPC efficiency (3.76%), which can in part be attributed to its performance advantage. One reason for decreased efficiency of the Linux VM is simply having to execute more instructions, such as additional VM Exits and extraneous kernel functionality (e.g., scheduling). Second, the additional interactions with the hypervisor and kernel on Linux increase the working set size and cause a 33% increase in LLC accesses. Third, Linux suffers nearly 9× more TLB misses than EbbRT. We attribute our TLB efficiency to our use of large pages throughout the system. Finally, we observe that even in a compute-bound task, the Linux VM spends a non-negligible amount of time in the kernel.

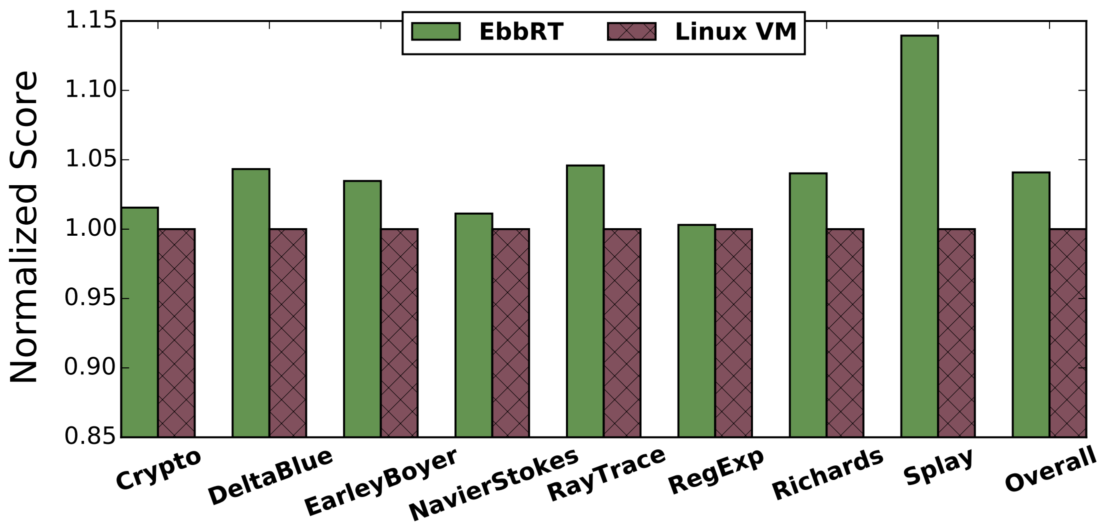


Figure 6.6: V8 JavaScript Benchmark

6.0.3.2 Node.js Webserver

Lastly, we evaluate a trivial webserver written for node.js, which uses the builtin `http` module and responds to each `GET` request with a small static message totaling 148 bytes. We use the `wrk`[39] benchmark to place moderate load on the webserver and measure mean and 99th percentile response-time latencies. EbbRT achieved 91.1 µs mean and 100.0 µs 99th percentile latencies. Linux achieved 103.5 µs mean and 120.6 µs 99th percentile latencies. The node.js webserver running on Linux has a 13.61% higher mean latency than the same webserver run on EbbRT. 99th percentile latency is 20.65% higher on Linux over EbbRT.

These results suggest that an entire class of server-side application written for node.js can achieve immediate performance advantages by simply running on top of EbbRT. Similar to our memcached evaluation, the ability for node.js to serve requests directly from hardware interrupts, without context switching or pre-emption, enables greater network performance. The non-preemptive run-to-completion execution model particularly improves tail latency. Our V8 benchmark results show that the use of large pages and simplified execution paths increases the efficiency of CPU and memory intensive workloads.

Finally, our approach opens up the application to further optimizations opportunities. For example, one could modify V8 to directly access the page tables to improve garbage collection[21]. We expect that greater performance can be achieved through continued system specialization.

Chapter 7

Conclusions and Future Work

As more businesses and users gain access to datacenter resources through cloud computing, resource efficiency becomes increasingly important. Stagnating CPU clock speeds and increasingly heterogeneous hardware makes it challenging for our existing general purpose operating systems to provide good performance for a diverse set of applications.

Rather than addressing the challenge of providing high performance in a general purpose operating system, this dissertation explores an alternative approach of providing high performance by constructing customized operating systems for individual applications. The challenge with this approach is enabling developers to build these custom operating systems with reasonable effort. To this end, we construct and evaluate a framework, the Elastic Building Block Runtime (EbbRT). EbbRT is described in detail through three sections:

- Chapter 3 describes the EbbRT execution environment. We discuss the non-preemptive, event-driven execution model, the identity mapped physical memory as well as the user-allocatable virtual memory, and finally the direct access to I/O devices, interrupts and DMA into application memory.
- Chapter 4 focuses on the programming primitives supported by EbbRT which promote the construction of reusable software. This chapter particularly focuses on the Elastic Building Block (Ebb) abstraction which encapsulates software components that can span both hosted and native execution environments. This chapter also discusses our use of lambdas, futures and IOBufs.
- Chapter 5 discusses how EbbRT provides compatibility with existing software. Specif-

ically, we describe the distinction between the hosted execution environment and the native execution environment and how EbbRT applications are constructed and deployed. We focus on the acceleration and offload enabled by this approach to either accelerate existing applications or rapidly develop new applications.

We evaluate EbbRT on two primary criteria: 1. The performance attainable by constructing custom library operating systems using EbbRT, and 2. The effort required to construct these library operating systems and the degree of reuse enabled by EbbRT. We evaluate the framework using a large collection of microbenchmarks and three different applications, memcached, node.js, and SageMath. This evaluation demonstrates that EbbRT library operating can significantly out perform existing, general purpose systems with only modest effort required.

7.0.1 Future Work

Much of the promise of EbbRT can only be truly evaluated as it scales to many developers and applications. Future work attempting to gain adoption of EbbRT would be valuable. EbbRT was developed over a number of years by a very small team (2 full-time, mostly junior developers at any one time). We explicitly avoided investing time and effort into aspects of the system that would have increased adoption (e.g. documentation, tutorials, tooling). Instead, we grappled with our own experience with the system and had more freedom to rapidly change the system; EbbRT was completely rewritten at least three times. The system has recently reached a level of maturity where it seems feasible to gain adoption beyond the small research group which built it. Often, adoption is looked at as an orthogonal property to other, more measurable, research properties such as performance, security, reliability, etc. However, we have argued throughout this dissertation that reducing developer effort is critical to the system success. Along with greater adoption comes a larger set of components that a developer can choose from and so adoption is an important property of a successful framework. Future work can explore which design decisions in EbbRT, while enabling high performance, can encourage adoption by a broader community.

In the same vein as achieving adoption, enabling a broader set of applications to be more easily deployed on EbbRT would have significant value. For example, a pthread and sockets implementation would allow non event-driven networking applications to be ported to EbbRT without modification. This may not provide the same degree of performance as the native EbbRT interfaces, but providing a path to optimization rather than wholesale redevelopment is important. Furthermore, providing tooling and guidance towards which portions of an application warrants optimizing would be an interesting direction for future work.

One of the least explored areas of the EbbRT research agenda has been the construction of distributed applications and the primitives required to support them. While Ebbs can span multiple machines and communicate through the use of Ebbs such as the `Messenger` and `GlobalIdMap`, these feel like very low-level primitives to construct a distributed component. Future work should look into ways to construct higher-level primitives and methods to construct distributed components.

7.0.2 Summary

Operating systems have historically been responsible for multiplexing the resources of a computer and providing abstractions to enable applications to be developed. The advent of cloud computing and the use of hardware virtualization has caused a renewed interest amongst the OS research community in library operating systems and other techniques which enable system functionality to be implenmented in userspace. We believe that for these techniques to be widely embraced, developers must be able to balance their desire to customize functionality with the need to reuse existing software. This dissertation has proposed a framework which allows developers to construct application-specific library operating systems. The Elastic Building Block Runtime has been demonstrated to improve performance of existing applications with relatively modest effort.

Bibliography

[1] Amazon Web Services (AWS) - Cloud Computing Services. `https://aws.amazon.com/`. Accessed: 2016-07-18.

[2] Boost.Asio. `http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio.html`.

[3] Cap'n Proto: Introduction. `http://capnproto.org/`. Accessed: 2016-07-18.

[4] Compute Engine - IaaS — Google Cloud Platform. `https://cloud.google.com/compute/`. Accessed: 2016-07-18.

[5] Docker. `https://www.docker.com/`.

[6] Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. `http://www.agner.org/optimize/instruction_tables.pdf`. Accessed: 2016-07-18.

[7] libuv. `http://libuv.org`.

[8] Multiboot Specification version 0.6.96. `https://www.gnu.org/software/grub/manual/multiboot/multiboot.html`. Accessed: 2016-07-18.

[9] One of the things I end up doing. `https://plus.google.com/+LinusTorvalds/posts/YDKRFDwHwr6`. Accessed: 2016-07-18.

[10] Sage. `http://www.sagemath.org`.

[11] Threading Building Blocks. `https://www.threadingbuildingblocks.org/`. Accessed: 2016-07-18.

[12] Userspace RCU. `http://liburcu.org/`. Accessed: 2016-07-18.

[13] Roce vs. iwarp competitive analysis. Technical report, 2015.

[14] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[15] Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, Eric Van Hensbergen, and Robert W. Wisniewski. Libra: A Library Operating System for a Jvm in a Virtualized Execution Environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 44–54, New York, NY, USA, 2007. ACM.

[16] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 95–109, New York, NY, USA, 1991. ACM.

[17] J. Appavoo, A. Waterland, and V. Uhlig. Project Kittyhawk: building a global-scale computer. *ACM SIGOPS Operating Systems Review*, 42(1):77, January 2008.

[18] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares. Experience distributing objects in an smmp os. *ACM Trans. Comput. Syst.*, 25(3), August 2007.

[19] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[20] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[21] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[22] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[23] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128. ACM, 2000.

[24] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing High-performance Memory Allocators. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 114–124. ACM, 2001.

[25] Brian N Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemysław Pardyak, Stefan Savage, and Emin Gün Sirer. SPIN - an extensible microkernel for application-specific operating system services. *ACM SIGOPS Operating Systems Review*, 29(1):74–77, 1995.

[26] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

[27] Georges Brun-Cottan and Mesaac Makpangou. Adaptable Replicated Objects in Distributed Environments. Research Report RR-2593, 1995. Projet SOR.

[28] Roy Campbell, Garry Johnston, and Vincent Russo. Choices (class hierarchical open interface for custom embedded systems). *ACM SIGOPS Operating Systems Review*, 21(3):9–17, 1987.

[29] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 66–77, New York, NY, USA, 2005. ACM.

[30] D. Chen, J. J. Parker, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, and B. Steinmacher-Burow. The IBM Blue Gene/Q Interconnection Network and Message Unit. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, page 1, 2011.

[31] Jonathan Corbet. SLQB - and then there were four. `http://lwn.net/Articles/311502`, Dec. 2008.

[32] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM.

[33] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[34] Jason Evans. jemalloc Tech Talk. `https://www.facebook.com/jemalloc/posts/189179837775115`, January 2011.

[35] Jason Evans. Scalable memory allocation using jemalloc. `http://www.canonware.com/jemalloc/`, 2011.

[36] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[37] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. *The Flux OSKit: A substrate for kernel and language research*, volume 31. ACM, 1997.

[38] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, 2009.

[39] Will Glozer. wrk: Modern HTTP benchmarking tool. `https://github.com/wg/wrk`, 2014.

[40] Google. Protocol Buffers: Google's Data Interchange Format. `https://developers.google.com/protocol-buffers`.

[41] Google. V8 Benchmark Suit - Version 7. `https://v8.googlecode.com/svn/data/benchmarks/v7/`.

[42] Google. V8 JavaScript Engine. `http://code.google.com/p/v8/`.

[43] Jason Hennessey, Sahil Tikale, Ata Turk, Emine Ugur Kaynar, Chris Hill, Peter Desnoyers, and Orran Krieger. Hil: Designing an exokernel for the data center. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 155–168, New York, NY, USA, 2016. ACM.

[44] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 13–26, New York, NY, USA, 1987. ACM.

[45] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[46] J. Jose, H. Subramoni, Miao Luo, Minjia Zhang, Jian Huang, M. Wasi-ur Rahman, N.S. Islam, Xiangyong Ouyang, Hao Wang, S. Sur, and D.K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 743–752, Sept 2011.

[47] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[48] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 9:1–9:14, New York, NY, USA, 2012. ACM.

[49] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. Technical report, Mountain View, CA, USA, 1994.

[50] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association.

[51] O. Krieger, P. McGachey, and A. Kanevsky. Enabling a marketplace of clouds: VMware's vCloud director. *SIGOPS Oper. Syst. Rev.*, 44:103–114, December 2010.

[52] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 133–145, New York, NY, USA, 2006. ACM.

[53] Orran Krieger, Orran Krieger, Michael Stumm, Michael Stumm, Ron Unrau, and Ron Unrau. The alloc stream facility: A redesign of application-level stream i/o. *IEEE Computer*, 27:75–82, 1992.

[54] Butler Lampson. An overview of the cal time-sharing system, 1969.

[55] Butler Lampson and Robert F Sproull. An Open Operating System for a Single-User Machine. Association for Computing Machinery, Inc., jan 1979.

[56] Jacob Leverich. Mutilate: High-Performance Memcached Load Generator. `https://github.com/leverich/mutilate`, 2014.

[57] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks*, pages 115–148. Ambient Intelligence. Springer Berlin Heidelberg, 2005.

[58] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[59] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[60] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 36–47, New York, NY, USA, 2013. ACM.

[61] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 260–267, New York, NY, USA, 1988. ACM.

[62] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. *SIGPLAN Not.*, 48(4):461–472, March 2013.

[63] Chris Maeda and Brian N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 244–255, New York, NY, USA, 1993. ACM.

[64] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented Objects for Distributed Abstractions. In Thomas L. Casavant and Mukesh Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, 1994.

[65] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM.

[66] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-Copy Update. In *Ottawa Linux Symposium*, July 2001. Available: `http://www.linuxsymposium.org/2001/abstracts/readcopy.php` `http://www.rdrop.com/users/paulmck/rclock/rclock_OLS.2001.05.01c.pdf` [Viewed June 23, 2004].

[67] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[68] Greg Minshall, Yasushi Saito, Jeffrey C. Mogul, and Ben Verghese. Application Performance Pitfalls and TCP's Nagle Algorithm. *SIGMETRICS Perform. Eval. Rev.*, 27(4):36–44, March 2000.

[69] José Moreira, Michael Brutman, José Castaños, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a Highly-Scalable Operating System: The Blue Gene/L story. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[70] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[71] Yoonho Park, Eric Van Hensbergen, Marius Hillenbrand, Todd Inglett, Bryan Rosenburg, Kyung Dong Ryu, and Robert W. Wisniewski. Fusedos: Fusing lwk performance with fwk functionality in a heterogeneous environment. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '12, pages 211–218, Washington, DC, USA, 2012. IEEE Computer Society.

[72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[73] Niels Provos and Nick Mathewson. libevent - an event notification library. `http://libevent.org/`, 2003.

[74] Injong Rhee, Nallathambi Balaguru, and George N Rouskas. MTCP: Scalable TCP-like congestion control for reliable multicast. *Computer networks*, 38(5):553–575, 2002.

[75] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system - assessment and perspectives. *Computing Systems*, 2:287–337, 1991.

[76] Quinn O Snell, Armin R Mikler, and John L Gustafson. Netpipe: A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, volume 6. Washington, DC, USA), 1996.

[77] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. `https://iperf.fr/`.

[78] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern linux api usage and compatibility: What to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 16:1–16:16, New York, NY, USA, 2016. ACM.

[79] S. Vinoski. Corba: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, Feb 1997.

[80] Rob von Behren, Jeremy Condit, and Eric Brewer. Why Events Are a Bad Idea (for High-concurrency Servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[81] Sara Williams and Charlie Kindel. The component object model: A technical overview. Technical report, Microsoft Technical Report, 1994.

[82] Robert W. Wisniewski, Todd Inglett, Pardo Keppel, Ravi Murty, and Rolf Riesen. mos: An architecture for extreme-scale operating systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '14, pages 2:1–2:8, New York, NY, USA, 2014. ACM.

Curriculum Vitae

[redacted]

[redacted]