EbbRT: A Framework for Building Per-Application Library Operating Systems

Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo, *Boston University*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/schatzberg

This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

EbbRT: A Framework for Building Per-Application Library Operating Systems

Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, Jonathan Appavoo
Boston University

Abstract

General purpose operating systems sacrifice per-application performance in order to preserve generality. On the other hand, substantial effort is required to customize or construct an operating system to meet the needs of an application. This paper describes the design and implementation of the Elastic Building Block Runtime (EbbRT), a *framework* for building per-application library operating systems. EbbRT reduces the effort required to construct and maintain library operating systems without hindering the degree of specialization required for high performance. We combine several techniques in order to achieve this, including a distributed OS architecture, a low-overhead component model, a lightweight event-driven runtime, and many language-level primitives. EbbRT is able to simultaneously enable performance specialization, support for a broad range of applications, and ease the burden of systems development.

An EbbRT prototype demonstrates the degree of customization made possible by our framework approach. In an evaluation of memcached, EbbRT and is able to attain 2.08× higher throughput than Linux. The node.js runtime, ported to EbbRT, demonstrates the broad applicability and ease of development enabled by our approach.

1 Introduction

Performance is a key concern for modern cloud applications. Even relatively small performance gains can result in significant cost savings at scale. The end of Dennard scaling and increasingly high-speed I/O devices has shifted the emphasis of performance to the CPU and, in turn, the software stack on which an application is deployed.

Existing general purpose operating systems sacrifice per-application performance in favor of generality. In response, there has been a renewed interest in library operating systems [28, 38], hardware virtualization [5, 44], and kernel bypass techniques [25, 48]. Common to these approaches is the desire to enable applications to interact with devices with minimal operating system involvement. This allows developers to customize the entire software stack to meet the needs of their application.

The problem with this approach is that it still requires significant engineering effort to implement the required system functionality. The consequence being that past systems have either targeted a narrow class of applications (e.g. packet processing) or, in order to be broadly applicable, constructed general purpose software stacks.

We believe that general purpose software stacks are subject to the same trade-off between performance and generality as existing commodity operating systems. In order to achieve high performance for a broad set of applications, we must bridge the gap between general purpose software, on which application development is easy, and the performance advantages obtained using customized, per-application software stacks where the development burden is high.

Our work addresses this gap with 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 combine several techniques in order to achieve this.

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 reuse of non-performance-critical components.

2. EbbRT uses a lightweight execution environment that allows application logic to directly interact with

hardware resources, such as memory and devices.

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

In this paper we describe the design and implementation of the EbbRT framework, along with several of its system components (e.g., a scalable network stack and a high-performance memory allocator). We demonstrate that library operating systems constructed using EbbRT outperform Linux on a series of compute and network intensive workloads. For example, a memcached port to EbbRT, run on a commodity hypervisor, is able to attain 2.08× greater throughput than memcached run on Linux. Additionally, we show that, with modest developer effort, EbbRT is able to support large, complex applications. In particular, node.js, a managed runtime environment, was ported in two weeks by a single developer.

The remainder of the paper is structured as follows: section 2 outlines the objectives of EbbRT, section 3 presents the high-level architecture and design of our framework, section 4 describes the implementation, section 5 presents the evaluation of EbbRT, section 6 discusses related work, and section 7 concludes.

2 Objectives

The following three objectives guide our design and implementation:

Performance Specialization: To achieve high performance we seek to allow applications to efficiently specialize the system at every level. To facilitate this, we provide an event-driven execution environment with minimal abstraction over the hardware; EbbRT applications can provide event handlers to directly serve hardware interrupts. Also, our Ebb component model has low enough overhead to be used throughout performance-sensitive paths, while also enabling compiler optimizations, such as aggressive inlining.

Broad Applicability: To ensure high utility, a framework should support a broad set of applications. EbbRT is designed and implemented to support the rich set of existing libraries and complex managed runtimes on which applications depend. We adopt a heterogeneous distributed architecture, called the MultiLibOS [49] model, wherein EbbRT library operating systems run alongside general purpose operating systems and offload functionality transparently. EbbRT library operating systems can be integrated with a process of the general purpose OS.

Ease of Development: We strive to make the development of application-specific systems easy. EbbRT exploits modern language techniques to simplify the task of writing new system software, while the Ebb model provides an abstraction to encapsulate existing system components. The barrier to porting existing applications is lowered through the use of function offloading between an EbbRT library OS and a general purpose OS.

Attaining all three of these objectives simultaneously is a challenging but critical step towards bridging the gap between general purpose and highly specialized software stacks.

3 System Design

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.

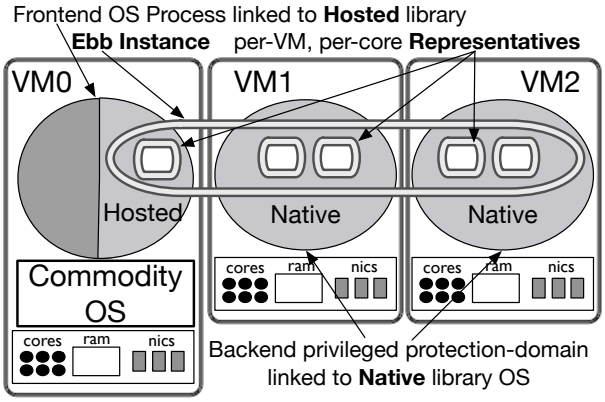

Figure 1: High Level EbbRT architecture

3.1 Heterogeneous Distributed Structure

Our design is motivated in-part by the common deployment strategies of cloud applications. Infrastructure as a Service enables 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 of a deployment. Rather, an application can be deployed across a heterogeneous mix of specialized library OSs and general purpose operating systems as illustrated in Figure 1.

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 [49]. 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.

3.2 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 [9, 39, 51], where the namespace of Ebbs is shared across both the native and hosted runtimes. Figure 1 illustrates an Ebb spanning both hosted and native runtimes of an application. 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.

3.3 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 [46], libuv [34], Boost ASIO [29]) to provide such an environment.

However, asynchronous, event-driven programming may not be a good fit for all applications. To this end, we provide a simple cooperative threading model on top of events. This allows for blocking semantics and a concurrency model similar to the Go programming language. We discuss support for long running events further in Section 4.2.

The non-preemptive event execution, along with support for cooperative threading, allows the native runtime to be lightweight yet provides sufficient flexibility for a wide range of applications. Such qualities are critical in enabling performance specialization without sacrificing applicability.

4 Implementation

In this section we provide an overview of the system software and then describe details of the implementation.

		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 IAAS resources
	Messenger	✓	✓		✓				Cross node Ebb to Ebb communication
	VirtioNet			✓	✓				VirtIO network device driver

Table 1: The core Ebbs that make up 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.

4.1 Software Structure Overview

EbbRT is comprised of an x86_64 library OS and toolchain as well as a Linux userspace library. Both runtimes are written predominately in C++14 totaling 14,577 lines of new code [59]. The native library is packaged with a GNU toolchain (gcc, binutils, libstdc++) and libc (newlib) modified to support a x86_64-ebbrt build target. Application code compiled with the toolchain will produce a bootable ELF binary linked with the library OS. We provide C and C++ standard library implementations which make it straightforward to use many third party software libraries as shown in Table 1. The support and use of standard software libraries to implement system-level functionality makes it much easier for library and application developers to understand and modify system-level Ebbs.

We chose not to strive for complete Linux or POSIX compatibility. We feel that enforcing compatibility with existing OS interfaces would be restrictive and, given the function offloading enabled by our heterogeneous distributed structure, unnecessary. Rather, we provide minimalist interfaces above the hardware, which allows for a broad set of software to be developed on top.

EbbRT provides the necessary functionality for events to execute and Ebbs to be constructed and used. This entails functionality such as memory management, networking, timers, and I/O. This functionality is provided by the core system Ebbs shown in Table 1

4.2 Events

Both the hosted and native environments provide an event driven execution model. Within the hosted environment we use the Boost ASIO library [29] in order to interface with the system APIs. Within the native environment, our event-driven API is implemented directly on top of the hardware interfaces. Here, we focus our description on the implementation of events within the native environment.

When the native environment boots, an event loop per core is initialized. Drivers 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. Therefore events are non-preemptive and typically generated by a hardware interrupt.

Applications can invoke synthetic events on any core in the system. The `Spawn` method of the

EventManager receives an event handler, which is later invoked from the event loop. Events invoked with Spawn are only executed once. A handler for a reoccurring event can be installed as an IdleHandler.

In order to prevent interrupt starvation, when an event completes the EventManager, 1. enables then disables interrupts, handling any pending interrupts, 2. dispatches a single synthetic event, 3. invokes all IdleHandlers and then 4. enables interrupts and halts. If any of these steps result in an event handler being invoked, then the process starts again at the beginning. This way, hardware interrupts and synthetic events are given priority over repeatedly invoked IdleHandlers.

While our EventManager implementation is simple, it provides sufficient functionality to achieve interesting dynamic behavior. For example, our network card driver implements 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 process each received packet to completion before returning control to the event loop. If the interrupt rate exceeds a configurable threshold, the driver disables the receive interrupt and installs an IdleHandler to process received packets. The EventManager will repeatedly call the idle handler from within the event loop, effectively polling the device for more data. When the packet arrival rate drops below a configurable threshold, the driver re-enables the receive interrupt and disables the idle handler, returning to interrupt-driven packet processing.

Given our desire to enable reuse of existing software, we adopt a cooperative threading model which allows events to explicitly save and restore control state (e.g., stack and volatile register state). At the point where the block would occur, the current event saves its state and relinquishes control back to the event loop, where the processing of pending events is resumed. The original event state can be restored, and its execution resumed, when the asynchronous work completes. The save and restore event mechanisms enable explicit cooperative scheduling between events, facilitating familiar blocking semantics. This has allowed us to quickly port software libraries that require blocking system calls.

A limitation of non-preemptive execution is the difficulty of mapping long-running threads with no I/O to an event-driven model. If the processor is not yielded periodically, event starvation can occur. At present we do not provide a completely satisfactory solution. Building a preemptive scheduler on top of events would be possible, though we fear it would fragment the set of Ebbs into those that depend on non-preemptive execution and those that don't. Alternatively, we have discussed dedicating processors to executing these long-running threads and therefore avoiding any starvation issues, similar to IX [5]. Nonetheless, we have not run into this problem in practice; most cloud applications rely heavily on I/O, and concern for starvation is reduced as we only support the execution of a single process.

4.3 Elastic Building Blocks

Nearly all software in EbbRT is written as elastic building blocks, which encapsulate both the data and function of a software component. Ebbs hide from clients the distributed or parallel nature of objects and can be extended or replaced for customization. An Ebb provides an interface using a standard C++ class definition. Every instance of an Ebb has a system-wide unique EbbId (32 bits in our current implementation). Software invokes the Ebb by converting the EbbId into an EbbRef which can be dereferenced to a per-core *representative* which is a reference to an instance of the underlying C++ class. We use C++ templates to implement the EbbRef generically for all Ebb classes.

Ebbs may be invoked on any machine or core within the application. Therefore, it is necessary for initialization of the per-core representatives to happen on-demand to mitigate initialization overheads for short-lived Ebbs. An EbbId provides an offset into a virtual memory region backed with distinct per-core pages which holds a pointer to the per-core representative (or NULL if it does not exist). When a function is called on an EbbRef, it checks the per-core representative pointer — in the common case where it is non-null, it is dereferenced and the call is made on the per-core representative. If the pointer is null, then a type specific *fault handler* is invoked which must return a reference to a representative to be called or throw a language-level exception. Typically, a fault handler will construct a representative and store it in the per-core virtual memory region so future invocations will take the fast-path. Our hosted implementation of Ebb dereferences uses per-thread hash-tables to store representative pointers.

The construction of a representative may require communication with other representatives either within the machine or on other machines. EbbRT provides core Ebbs that support distributed data storage and messaging services. These facilities span and enable communication between the EbbRT native and hosted instances and utilize network communication as needed.

Ebb modularity is both flexible and efficient, making them suitable for high-performance components. Previous systems providing a partitioned object model either used relatively heavy weight invocation across a distributed system [56], or more efficient techniques constrained to a shared memory system [17, 30]. Ebbs are unique in their ability to accommodate both use cases.

The fast-path cost of an Ebb invocation is one predictable conditional branch and one unconditional branch more than a normal C++ object dereference. Additionally, our use of static dispatch (`EbbRef`'s are templated by the representative's type) enables compiler optimizations such as function inlining.

We intentionally avoided using interface definition languages such as COM [60], CORBA [57], or Protocol Buffers [21]. Our concern was that these often require serialization and deserialization at all interface boundaries, which would promote much coarser grained objects than we desire. Our ability to use native C++ interfaces allows Ebbs to pass complex data structures amongst each other. This also necessitates that all Ebb invocations be local. Ebb representative communication is encapsulated and may internally serialize data structures as needed.

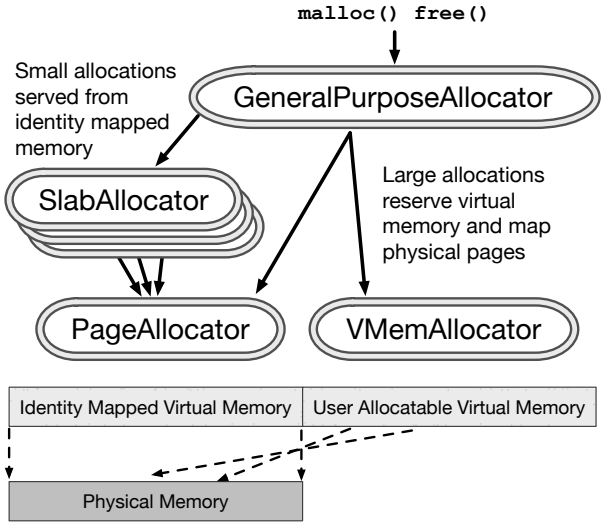

Figure 2: Memory Management Ebbs

4.4 Memory Management

Memory allocation is a performance critical facet of many cloud applications, and our focus on short-lived events puts increased pressure on the memory allocator to perform well. Here we present our default native memory allocator and highlight aspects of it which demonstrate the synergy of the elements in the EbbRT design. Figure 2 illustrates EbbRT's memory management infrastructure and the virtual and physical address space relationships.

The EbbRT memory allocation subsystem is similar to that of the Linux Kernel. The lowest-level allocator is the `PageAllocator`, which allocates power of two sized pages of memory. Our default `PageAllocator` implementation uses buddy-allocators, one per NUMA node. On top of the `PageAllocator` are the `SlabAllocator` Ebbs, which are used to allocate fixed size objects. Our default `SlabAllocator` implementation uses per-core and per-NUMA node representatives to store object free-lists and partially allocated pages. This design is based on the Linux Kernel's SLQB allocator [12]. The `GeneralPurposeAllocator`, invoked via malloc, is implemented using multiple `SlabAllocator` instances, each responsible for allocating objects of a different fixed size. To serve a request, the `GeneralPurposeAllocator` invokes the `SlabAllocator` with the closest size greater or equal to the requested size. For requests exceeding the largest `SlabAllocator` size, the `GeneralPurposeAllocator` will allocate a virtual memory region mapped in from the `VMemAllocator` and backed by pages mapped in from the `PageAllocator`.

By defining the memory allocators as Ebbs, we allow any one of the components to be replaced or modified without impacting the others. In addition, because our implementation uses C++ templates for static dispatch, the compiler is able to optimize calls across Ebb interfaces. For example, calls to `malloc` that pass a size known at compile time are optimized to directly invoke the correct `SlabAllocator` within the `GeneralPurposeAllocator`.

A key property of memory allocations in EbbRT is that most allocations are serviced from identity mapped physical memory. This applies to all allocations made by the `GeneralPurposeAllocator` that do not exceed the largest `SlabAllocator` size (virtual mappings are used for larger allocations). Identity mapped memory allows application software to perform zero-copy I/O with standard allocations rather than needing to allocate memory specifically for DMA.

Another benefit of the EbbRT design is that, due to the lack of preemption, most allocations can be serviced from a per-core cache without any synchronization. Avoiding atomic operations is so important that high performance allocators like TCMalloc [19] and jemalloc [14] use per-*thread* caches to do so. These allocators then require complicated algorithms to balance the caching across a potentially dynamic set of threads. In contrast, the number of cores is typically static and generally not too large, simplifying EbbRT's balancing algorithm.

While a portion of the virtual address space is reserved to identity map physical memory and some virtual memory is used to provide per-core regions for Ebb invocation, the vast majority of the virtual address space is available for application use. Applications can allocate virtual regions by invoking the `VMemAllocator` and passing in a handler to be invoked on faults to that allocated region. This allows applications to implement

```
1  // Sends out an IPv4 packet over Ethernet
2  Future<void> EthIpv4Send(uint16_t eth_proto, const Ipv4Header& ip_hdr, IOBuf buf) {
3    Ipv4Address local_dest = Route(ip_hdr.dst);
4    Future<EthAddr> future_macaddr = ArpFind(local_dest);    /* asynchronous call   */
5    return future_macaddr.Then(
6      // continuation is passed in as an argument
7      [buf = move(buf), eth_proto](Future<EthAddr> f) {      /* lambda definition   */
8        auto& eth_hdr = buf->Get<EthernetHeader>();
9        eth_hdr.dst = f.Get();
10       eth_hdr.src = Address();
11       eth_hdr.type = htons(eth_proto);
12       Send(move(buf));
13     });                                                    /* end of Then() call  */
14 }
```

Figure 3: Network code path to route and send and Ethernet frame.

arbitrary paging policies.

Our memory management demonstrates some of the advantages provided by EbbRT's design. First, we use Ebbs to create per-core representatives for multi-core scalability and to provide encapsulation to enable the different allocators to be replaced. Second, the lack of preemption enables us to use the per-core representatives without synchronization. Third, the library OS design enables tighter collaboration between system components and application components — as exemplified by the application's ability to directly manage virtual memory regions and achieve zero-copy interactions with device code.

4.5 Lambdas and Futures

One of the core objectives of our design is mitigating complexity to ease development. 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 [58]. 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 referred to 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. Because event-driven programming splits one logical flow of control across multiple stacks, exceptions must be handled at every event boundary. This puts the 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 these problems is our implementation of *monadic futures*. Futures are a data type for asynchronously produced values, originally developed for use in the construction of distributed systems [37]. Figure 3 illustrates a code path in the EbbRT network stack that utilizes lambdas and futures to route and send an Ethernet frame. The `ArpFind` function (line 4) translates an IP address to the corresponding MAC address either through a lookup into the ARP cache or by sending out an ARP request to be processed asynchronously. In either case, `ArpFind` returns a `Future<EthAddr>`, which represents the future result of the ARP translation. A future cannot be directly operated on. Instead, a lambda can be applied to it using the `Then` method (line 5). This lambda is invoked once the future is fulfilled. When invoked, the lambda receives the fulfilled future as a parameter and can use the `Get` method to retrieve its underlying value (line 9). In the event that the future is fulfilled before the `Then` method is invoked (for example, `ArpFind` retrieves the translation directly from the ARP cache) the lambda 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 `EthIpv4Send` method returns a `Future<void>` which merely rep-

resents 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 `EthIpv4Send` and handled by higher-level code.

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 provide monadic futures similar to ours.

As seen in Table 1, futures are used pervasively in interface definitions for Ebbs, 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.

4.6 Network Stack

We originally looked into porting an existing network stack to EbbRT. However, we eventually implemented a new network stack for the native environment, providing IPv4, UDP/TCP, and DHCP functionality in order to provide an event-driven interface to applications, minimize multi-core synchronization, and enable 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.

During the development of EbbRT we found it necessary to create a common primitive for managing data that could be received from or sent to hardware devices. To support the development of zero-copy software, we created the `IOBuf` primitive. An `IOBuf` is a descriptor which manages ownership of a region of memory as well as a *view* of a portion of that memory. Rather than having applications explicitly invoke `read` with a buffer to be populated, they install a handler which is passed an `IOBuf` containing network data for their connection. This `IOBuf` is passed synchronously from the device driver through the network stack. The network stack does not provide any buffering, it will invoke the application as long as data arrives. Likewise, the interface to send data accepts a chain of `IOBufs` which can use scatter/gather interfaces.

Most systems have fixed size 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 [41]. 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 [47]. EbbRT stores connection state in an RCU [40] hash table which allows common connection lookup operations to proceed without any atomic operations. Due to the event-driven execution model of EbbRT, RCU is a natural primitive to provide. Because we lack preemption, entering and exiting RCU critical sections are free. 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 network stack is an example of the degree of performance specialization our design enables. By involving the application in network resource management, the networking stack avoids significant complexity. 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.

5 Evaluation

Through evaluating EbbRT we aim to affirm that our implementation fulfills the following three objectives discussed in Section 2: 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 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 evaluation on the EbbRT native environment.

5.1 Microbenchmarks

The first 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 second 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.

5.1.1 Memory Allocation

In K42 [30], we 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 [7]. We've ported *threadtest* from the Hoard [6] 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.

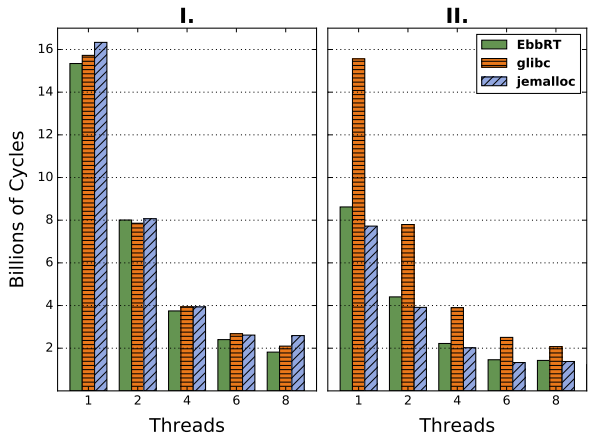

Figure 4: Hoard Threadtest. Y-axis represents threads, t. I.) N=100,000, i=1000; II.) N=100, i=1,000,000.

In *threadtest*, each thread t allocates and frees $\frac{N}{t}$ 8 byte objects. This task is repeated for i iterations. Figure 4 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.

5.1.2 Network Stack

To evaluate the performance of our network stack we ported the NetPIPE [52] and iPerf [54] 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 5 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.

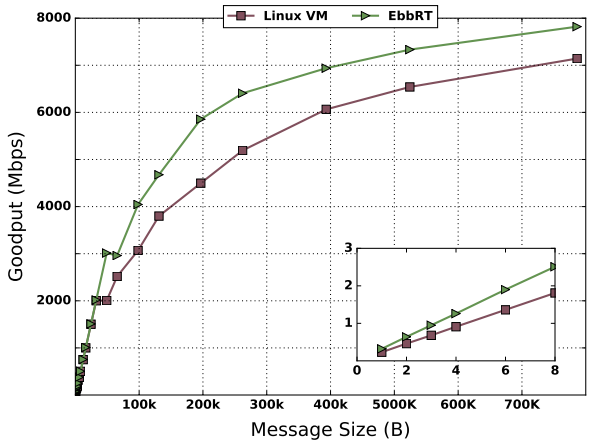

Figure 5: NetPIPE performance as a function of message size. Inset shows small message sizes.

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.

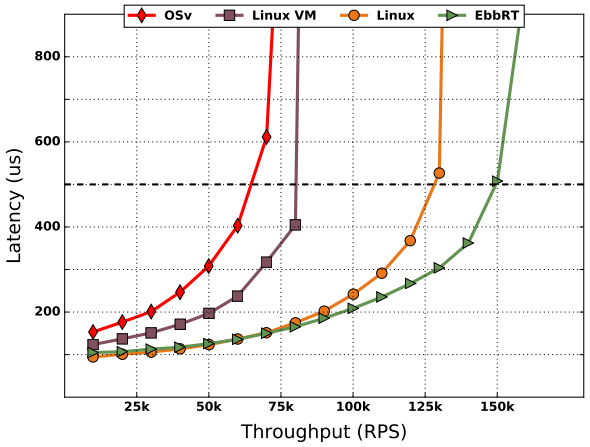

Figure 6: Memcached Single Core Performance

This explains the difference in Netpipe goodput before the network becomes the bottleneck.

5.2 Memcached

We evaluate memcached [15], 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 [27], 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 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` [31] benchmarking tool to place a particular load on the server and measure response latency. We configure

	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 2: Memcached CPU-efficiency metrics

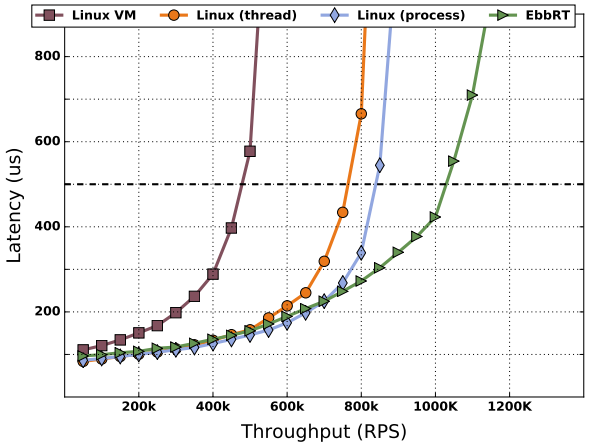

Figure 7: Memcached Multicore Performance

`mutilate` to generate load representative of the Facebook ETC workload [2], 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 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 [28], 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 7) due to a lack of multiqueue support in their `virtio-net` device driver.

Figure 7 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 7). 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 2 presents these statistics. We see that the EbbRT server is processing requests 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 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 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 ap-

	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 3: Memcached Per-Request Latency

plication 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 [35, 43] or the use of custom hardware [26, 36]. By writing memcached as an EbbRT application, we are able to achieve significant performance improvements while maintaining compatibility with standard clients, protocols, and hardware.

5.3 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 [23], Google's JavaScript engine, and libuv [34], 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.

5.3.1 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 [22]. This collection of purely compute-bound benchmarks stresses the core performance of the V8 JavaScript engine. Figure 8 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 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 effi-

	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 4: V8 JavaScript Benchmark CPU-efficiency metrics

ciency 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.

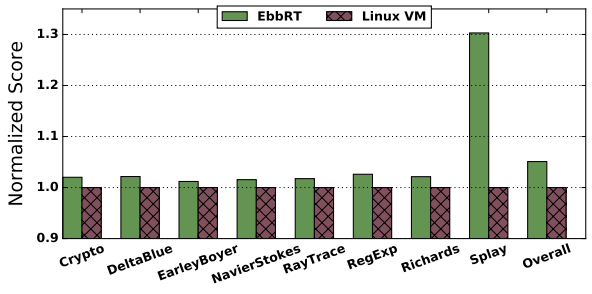

Figure 8: V8 JavaScript Benchmark

5.3.2 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` [20] 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 [4]. We expect that greater performance can be achieved through continued system specialization.

6 Related Work

The Exokernel [13] introduced the library operating system structure — where system functionality is directly linked into the application and executes in the same address space and protection domain. Library operating systems have been shown to provide many useful properties, such as portability [45, 55], security [3, 38], and efficiency [28]. OSv [28] and Mirage [38] are similar to EbbRT in that they target virtual machines deployed in IaaS clouds. OSv constructs a general purpose library OS and supports the Linux ABI. Mirage uses the OCaml programming language to construct minimal systems for security. EbbRT takes a middle ground, supporting source-level portability for existing applications through rich C++ functionality and standard libraries, but avoiding general purpose OS interfaces.

CNK [42], Libra [1], Azul [53], and, more recently, Arrakis [44] and IX [5] have pursued architectures that enable specialized execution environments for performance sensitive data flow. While their approaches vary, these systems must each make a trade-off between targeting a narrow class of applications (e.g., HPC, Java web applications, or packet processing) and targeting a broad class of applications. Rather than supporting a single specialized execution environment, EbbRT provides a framework to enable the construction of various application-specific library operating systems.

Considerable work has been done on system software customization [8, 11, 17, 30, 50]. Much of this work focuses on developing general purpose operating that are customizable, while EbbRT is focused on the construction of specialized systems.

Choices [10] and OSKit [16] provide operating system frameworks; in the case of Choices, for maintainability and extensibility, and in the case of OSKit, to simplify the construction of new operating systems. EbbRT differs most significantly in its performance objectives and its focus on enabling application developers to extend and customize system software. For example, by pro-

viding EbbRT as a modified toolchain, application-level software libraries (e.g. boost) can be used in a systems context with little to no modification.

Others have considered the interaction of an object-oriented framework with the goal of enabling high performance. CHAOSarc [18] and TinyOS [32] have both explored the use of a fine grain object framework in the resource limited setting of embedded systems. Like TinyOS, EbbRT combines language support for event driven execution and method dispatch mechanisms that are friendly to compiler optimization. In a recent retrospective [33], the authors recognized that their development and use of the nesC programming language limited TinyOS from even broader, long-term use. EbbRT, however, focuses on integration with existing software tooling, development patterns, and libraries to encourage continued applicability.

7 Concluding Remarks

We have presented EbbRT, a framework for constructing specialized systems for cloud applications. Through our evaluation we have established that EbbRT applications achieve their performance advantages through system-wide specialization rather than one particular technique. In addition, we have shown that existing applications can be ported to EbbRT with modest effort and achieve a noticeable performance gain. Throughout this paper we have conveyed how our primary design elements, i.e., elastic building blocks, an event-driven execution environment, and a heterogeneous deployment model working alongside advanced language constructs and new system primitives, have proven to be a novel, effective approach to achieving our stated objectives.

By encapsulating system components with minimal dispatch overhead, we enable application-specific performance specialization throughout all parts of our system. Furthermore, we have shown that our default Ebb implementations provide a foundation for achieving performance advantages. For example, our results illustrate the combined benefits of using a non-preemptive event-driven execution model, identity mapped memory, and zero-copy paths.

As we gained experience with the system the issue of easing development efforts arose naturally. An early focus on enabling use of standard libraries, including the addition of blocking primitives, greatly simplified development. Our monadic futures implementation addressed concrete concerns we had with event-driven programming. Futures are now used throughout our implementation. IOBufs came about as a solution for us to enable pervasive zero-copy with little added complexity or overhead.

EbbRT's long-term utility hinges on its ability to be used for a broad range of applications, while continuing to enable a high degree of per-application specialization. Our previous work on fragmented objects gives us confidence that various specialized implementations of our existing Ebbs can be introduced, as needed, without diminishing the overall value and integrity of the framework.

Future work involves further exploration of system specialization. Our focus has primarily revolved around networked applications, however, data storage applications should equally benefit from specialization. Additionally, EbbRT can be used to accelerate existing applications in a fine-grained fashion. We believe the hosted library can be used not just for compatibility for new applications, but as a way to offload performance critical functionality to one or more library operating systems.

The EbbRT framework is open source and actively used in ongoing systems research. We invite developers and researchers alike to visit our online codebase at `https://github.com/sesa/ebbrt/`

Acknowledgments: We owe a great deal of gratitude to the Massachusetts Open Cloud (MOC) team for their help and support in getting EbbRT running and debugged on HIL [24]. We would like to thank Michael Stumm, Larry Rudolph and Frank Bellosa for feedback on early drafts of this paper. Thanks to Tommy Unger, Kyle Hogan and David Yeung for helping us get the nits out. We would also like to thank our shepherd, Andrew Baumann, for his help in preparing the final version. This work was supported by the National Science Foundation under award IDs CNS-1012798, CNS-1254029, CCF-1533663 and CNS-1414119. Early work was supported by by the Department of Energy under Award Number DE-SC0005365.

References

[1] 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. ACM, 2007.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64. ACM, 2012.

[3] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, August 2015.

[4] 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. USENIX Association, 2012.

[5] 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.

[6] 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.

[7] 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.

[8] 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. *SIGOPS Oper. Syst. Rev.*, 29(1):74–77, January 1995.

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

[10] Roy H. Campbell, Nayeem Islam, and Peter Madany. Choices, frameworks and refinement. *Computing Systems*, 5(3):217–257, 1992.

[11] David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94. USENIX Association, 1994.

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

[13] 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. ACM, 1995.

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

[15] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5, August 2004.

[16] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 38–51. ACM, 1997.

[17] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 87–100. USENIX Association, 1999.

[18] Ahmed Gheith and Karsten Schwan. CHAOSarc: Kernel Support for Multiweight Objects, Invocations, and Atomicity in Real-time Multiprocessor Applications. *ACM Trans. Comput. Syst.*, 11(1):33–72, February 1993.

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

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

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

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

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

[24] 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*, pages 155–168. ACM, 2016.

[25] Intel Corporation. Intel DPDK: Data Plane Development Kit. `http://dpdk.org`.

[26] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 743–752. IEEE Computer Society, 2011.

[27] 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. ACM, 2012.

[28] 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. USENIX Association, June 2014.

[29] Kohlhoff, Christopher. Boost.Asio. `http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio.html`.

[30] 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. ACM, 2006.

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

[32] 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.

[33] Philip Levis. Experiences from a Decade of TinyOS Development. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 207–220. USENIX Association, 2012.

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

[35] 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. USENIX Association, April 2014.

[36] 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. ACM, 2013.

[37] Barbara Liskov and Liuba 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. ACM, 1988.

[38] 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.

[39] Mesaac Makpangou, Yvon Gourhant, and Jean pierre Le Narzul. Fragmented Objects for Distributed Abstractions. In *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, 1992.

[40] 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.

[41] 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.

[42] 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. ACM, 2006.

[43] 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 *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 385–398. USENIX Association, 2013.

[44] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16. USENIX Association, October 2014.

[45] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304. ACM, 2011.

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

[47] Injong Rhee, Nallathambi Balaguru, and George N. Rouskas. MTCP: Scalable TCP-like Congestion Control for Reliable Multicast. *Comput. Netw.*, 38(5):553–575, April 2002.

[48] Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 9. USENIX Association, 2012.

[49] Dan Schatzberg, James Cadden, Orran Krieger, and Jonathan Appavoo. A Way Forward: Enabling Operating System Innovation in the Cloud. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*. USENIX Association, June 2014.

[50] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 213–227, New York, NY, USA, 1996. ACM.

[51] 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.

[52] 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*, 1996.

[53] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88. ACM, 2011.

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

[55] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation

and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 9:1–9:14. ACM, 2014.

[56] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78, January 1999.

[57] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *Comm. Mag.*, 35(2):46–55, February 1997.

[58] 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, page 4. USENIX Association, 2003.

[59] David A Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.

[60] Sara Williams and Charlie Kindel. The Component Object Model: A Technical Overview. *Dr. Dobbs Journal*, 356:356–375, 1994.