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

Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, Jonathan Appavoo
Department of Computer Science, Boston University

Efficient use of high speed hardware requires operating system components be customized to the application workload. Our general purpose operating systems are ill-suited for this task. We present EbbRT, a framework for constructing per-application library operating systems for cloud applications. The primary objective of EbbRT is to enable high-performance in a tractable and maintainable fashion. This paper describes the design and implementation of EbbRT, and evaluates its ability to improve the performance of common cloud applications. The evaluation of the EbbRT prototype demonstrates memcached, run within a VM, can outperform memcached run on an unvirtualized Linux. The prototype evaluation also demonstrates an 14% *performance improvement of a V8 JavaScript engine benchmark, and a node.js webserver that achieves a 50% reduction in* 99*th percentile latency compared to it run on Linux.*

1. Introduction

Conventional wisdom is that application performance can be increased through operating system specialization. There has been a renewed interest in library operating systems [33], hardware virtualization [8, 37], and kernel bypass techniques [24, 41]. Common in these approaches is the desire to enable applications to directly manage virtual devices with minimal operating system involvement. This allows developers to customize the entire software stack to the needs of their application. We believe that such customization will prove necessary to fully exploit the potential of advanced datacenter hardware, such as high speed networking and low latency storage.

One of the critical challenges of these approaches is how to make the degree of per-application customization tractable and maintainable. This paper describes the design and implementation of the Elastic Building Block Runtime (EbbRT), a framework for constructing maintainable, high-performance library operating systems. We combine several techniques in order to achieve this:

1. EbbRT is comprised of a set of components that developers can extend, replace or discard in order to construct and deploy a particular application. This enables a much greater degree of customization than existing general purpose systems while promoting the reuse of non-performance-critical components.
2. EbbRT components run in a light-weight event-driven environment. This reduces the run-time complexity yet provides enough flexibility for a wide range of applications.
3. EbbRT library operating systems can run within virtual machines on unmodified hypervisors. This allows us to deploy EbbRT applications on commodity clouds.
4. EbbRT library operating systems run alongside general purpose operating systems. This allows functionality to be offloaded for compatibility, reducing the maintenance burden by avoiding the construction of new software.
5. EbbRT uses many modern and high-level programming techniques not typically found in operating systems software. This was chosen deliberately to reduce the complexity of the software.

In this paper, we demonstrate that EbbRT library operating systems outperform Linux on a series of compute and network intensive workloads. For example, a memcached port to EbbRT, run within a commodity hypervisor, is able to attain 58% greater throughput than memcached run within a Linux virtual machine and is able surpass the performance of memcached running directly on the native Linux host. Additionally, we demonstrate that EbbRT is able to support a managed runtime environment, node.js, with modest developer effort. Our port of this environment is able to attain a 13.9% improvement on a standard javascript benchmark and over 50% improvement in 99th percentile latency of a node.js webserver.

2. System Design

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

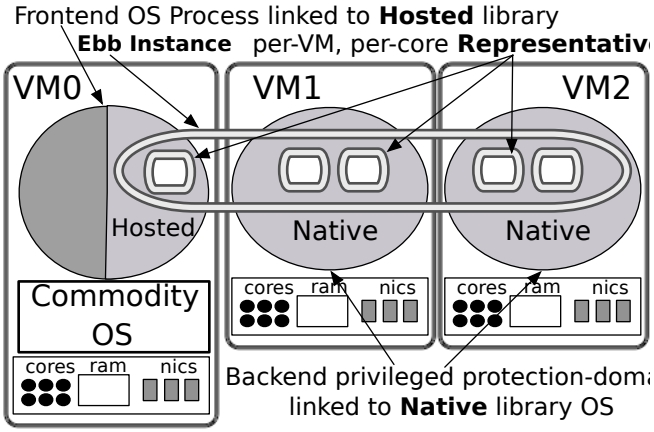

Figure 1: High Level EbbRT architecture

2.1. Heterogeneous Distributed Structure

Our design is motivated by the common deployment strategies of cloud applications. Infrastructure as a Service providers

enable a single application to be deployed across multiple machines within an isolated network. Within this context, there is no need to run general purpose operating systems on all machines of an application, instead we employ a heterogeneous mixture of general purpose operating systems and specialized library OSs as illustrated in Figure 1.

EbbRT consists of a small runtime and a collection of software components. One implementation of the runtime is distributed with an associated toolchain which constructs bootable application binaries. We refer to this as a *native* environment. We also support another implementation which can be embedded into a process of a general purpose OS by linking to a user-level library which we refer to as a *hosted* environment.

A common deployment of a EbbRT application launches a hosted process and one or more native library OSs which communicate via a local network. A user is able to interact with the EbbRT application as they would any other process of the general purpose OS. A EbbRT application may provide additional interfaces to the native portions of the application, or offload functionality transparently.

The native library OS 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 environment sets up a single address space, basic system functionality (e.g. timers, networking, memory allocation) and invokes an application entry point while still running at the highest privilege level.

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 RPCs to the hosted environment. This is critical in enabling a light-weight high-performance native environment. Furthermore, our experience has been that developing operating systems is largely dominated by providing compatibility with legacy interfaces. In EbbRT, we opted to reuse existing systems for that purpose. The most maintainable software is that which was not written.

This structure is by no means set in stone. Applications can be deployed with purely native library operating systems or purely hosted applications. However, we feel that the strength of the system is in the ability to span both simultaneously. Our design also allows EbbRT library operating systems to run directly on hardware (without virtualization). In either case, EbbRT depends on application isolation at the network layer, either through switch programming or virtualization.

2.2. Modular System Structure

EbbRT allows application developers to modify or extend all levels of the initial EbbRT environment. In order to support this, a EbbRT application is almost entirely comprised of objects we call *Elastic Building Blocks* (Ebbs). As with objects in many programming languages, they encapsulate implementation details behind a well-defined interface.

The namespace of Ebbs are shared across all machines in the system (hosted and native). Ebbs are distributed, multi-core fragmented objects [10, 34, 42]. 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 such as RDMA, caching, and using local storage, etc. would all be hidden from clients of the filesystem Ebb.

Motivated by the desire to enable even the lowest levels of the native environment to be modular, Ebbs must also 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.

In the context of our goal of maintainability, this approach promotes the creation of reusable software. When more developers depend on the functionality of any one Ebb, the individual burden of maintaining it is reduced.

2.3. Non-preemptive Event-driven Execution

Execution in EbbRT is non-preemptive. There is one event loop per-core which dispatches external events, such as timer completions or device interrupts, and software generated events. A registered event handler is invoked which runs without preemption. This model is in contrast to a more standard threaded environment where multiple threads are multiplexed across one or more cores with preemption. Our event-driven execution environment can be efficiently implemented directly on top of interrupts, providing a low overhead abstraction over the hardware. This allows application software to run immediately off a device interrupt without the typical costs of scheduling decisions or protection domain switches.

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

Many cloud applications are driven by external requests such as network traffic and 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[38], libuv[4], Boost ASIO[2]) to provide such an environment. However, this 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.

This environment allows the run-time to be light-weight yet

provides sufficient flexibility for a wide range of applications. This enables greater reuse of a smaller codebase and allows customization to occur throughout the entire software stack.

3. Implementation

In this section we present salient aspects of the EbbRT implementation. We first highlight the challenges that the implementation must address given the goals and design of the system. We then briefly provide an overview of the software structure and describe the details of the implementation.

Inherent to our goal of increased customizability is the potential for greater complexity for developers. We rely heavily on the Elastic Building Block model to mitigate this. However, this is only effective so long as components can be reused. This presents several challenges:

1. The implementation of Ebbs must permit fine-grained decomposition without loss of performance. Otherwise, developers will be pushed to construct large, monolithic structures will little room for customization.
2. The interfaces of Ebbs must provide the flexibility for a broad set of implementations and also promote interoperability amongst components. Therefore, our choice of common primitives is critical.
3. The native runtime must not inhibit an application developer from fully exploiting the functionality of the hardware. Otherwise, the developer community will fragment.

3.1. Software Structure Overview

EbbRT is comprised of a set of an x86_64 library OS and toolchain as well as a Linux userspace library. Both runtimes are written predominately in C++11 totaling 13,930 lines of new code[48]. The native library OS is deployed along with a modified GNU toolchain (gcc, binutils, libstdc++) and newlib (libc) that provide an x86_64-ebbrt target. Application code targeting the native library OS is compiled with this toolchain and the resulting binary is a bootable ELF linked with the library OS. We provide C and C++ standard library implementations which make it straightforward to use many third party software libraries. For example, we use many of the Boost C++ libraries [1] within our system.

The EbbRT runtime provides the minimum necessary functionality for events to execute and Ebbs to be constructed and used. This entails functionality such as memory management and allocation, networking, timers, and interrupt control. This functionality is provided by several Ebbs. Every EbbRT library OS must be deployed with some implementation of these Ebbs. We provide a default implementation, though they can be replaced.

We do not strive for Linux ABI compatibility or POSIX API compatibility. Rather, we provide minimal interfaces above the hardware to enable the broadest set of software to be developed on top. We feel that enforcing compatibility with existing OS interfaces would be restrictive.

3.2. Events

Both the hosted and native environments provide an event driven execution model. Within the hosted environment we use the Boost ASIO library[2] 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. While a processor is executing an event, all interrupts are disabled. When an event completes, interrupts are enabled and more events can be processed. Therefore events are non-preemptive and typically generated by a hardware interrupt. 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 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, if one exists; 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 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 each 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 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 our `EventManager` implementation is simple, it provides sufficient functionality to implement this dynamic behavior.

A common challenge associated with event-driven 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.[5] refer to this as *stack ripping*. Given our desire to enable reuse of existing software, we use a hybrid model that allows events to explicitly save and restore event state (the stack and volatile register state). This has allowed us to quickly port software libraries which require a typically blocking system call. At the point where the block would occur, the current event saves its state and 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 in order to provide familiar blocking semantics.

As we will discuss later, many components of EbbRT 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 [13]. 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.

A limitation of this 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. 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.

3.3. Elastic Building Blocks

Nearly all software in EbbRT is written as Elastic Building Blocks (Ebbs) which encapsulate both the data and function of a software component. An Ebb provides a functional 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* and 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. In the case that an Ebb is short-lived and only accessed on one core, initializing representatives aggressively would incur significant overhead. 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. Due to the lack of per-core virtual memory regions available in Linux userspace, our hosted implementation relies on per-core hash-tables to store representative pointers.

To construct a representative may require communication with other representatives either within the machine or on other machines. EbbRT provides additional Ebbs that support distributed data storage, messaging, naming and location services. These facilities span and enable communication between the EbbRT native and hosted instances and utilize network communication as needed. In this paper we focus primarily on the per-machine structure and therefore we omit further discussion of these Ebbs in this paper.

Ebbs are both flexible and efficient. Previous systems providing a partitioned object model either used relatively heavy weight invocation across a distributed system, or more efficient techniques constrained to a shared memory system. 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. This makes Ebbs suitable for components with high-performance demands such as the memory allocator.

We intentionally avoided using interface definition languages such as COM [49], CORBA [47], or Protocol Buffers [22]. Our concern was that these often require serialization and deserialization at interface boundaries. This 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 invocation be local. If Ebb representatives wish to communicate, then they may internally serialize data over the network, though this is hidden from the Ebb clients.

3.4. Memory Allocation

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. We highlight a number of aspects of the allocator which demonstrate the synergy of the EbbRT design.

The EbbRT memory allocation subsystem is similar to that

of Linux. The lowest-level allocator is the page allocator Ebb which allocates power of two sized pages of memory. Our default implementation uses per-numa-node buddy-allocators. On top of the page allocator are slab allocators which can be used to allocate fixed size objects. Our default slab allocator uses per-core and per-numa-node representatives to store object free-lists and partially allocated pages. This design is based on Linux's SLQB allocator [12]. The general purpose memory allocator, invoked via malloc, is implemented using many slab allocators, each allocating objects of different sizes. To serve a request, the slab allocator with the closest size greater or equal to the requested size is invoked. Allocations larger than the largest slab allocator size instead allocate a virtual memory region and map in pages from the page allocator.

All three allocators are defined using Ebbs which allow any one of the components to be replaced or modified without impacting the others. In previous systems [28], the overhead of a partitioned object model prevented its use for high performance components such as the memory allocator. In contrast, Ebbs are lightweight enough to be usable by our memory allocator. In fact, because our implementation uses C++ templates for static dispatch, the compiler is able to optimize calls across an Ebb interface, something that previous partitioned object models prevented. Most calls to malloc pass a size which is known at compile time. We noticed that these calls were being optimized to directly invoke the correct slab allocator within the general purpose allocator.

The same general purpose allocator is used for system level allocations (e.g. networking data structures) as well as for application level allocations. A key property of the allocator is that, except for very large allocations, allocations are serviced from identity mapped physical memory. This allows application software to perform zero-copy I/O with data allocated from the standard memory allocator rather than needing to allocate memory specifically for DMA.

Another property of the allocator 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[20] and jemalloc[16] 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 some virtual memory is reserved to identity map the physical memory and 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 and provide their own page fault handler which is invoked on faults to that region. This allows applications to implement arbitrary paging policies.

Our memory allocator demonstrates some of the advantages provided by EbbRT's design. First, we use Ebbs to create per-core representatives for multi-core scalability and also to provide encapsulation to enable the different allocators to be replaced. Second, the use of non-preemptive events enables us to use the per-core representatives without synchronization. And third, the library OS design enables tighter collaboration between system components and application components - as exemplified by the page allocator communicating memory pressure up to higher-level caches.

3.5. Lambdas and Futures

One of the core principles of our design is mitigating complexity. 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. Tasks usually invoke a function, passing in an event handler to be invoked upon completion of some future task (e.g. I/O). The event handler is invoked within a different context than the original function was invoked and so it falls on the programmer to manually save and restore state across invocations.

A new C++ feature, lambdas, allow programmers to define anonymous functions inline. Lambdas can also capture local state so that it can be referred to when the lambda is invoked. This removes the burden of manually saving and restoring state and also makes code easier to follow. We use lambdas ubiquitously in EbbRT to construct 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 *monadic futures*. Figure 2 illustrates a code path in the EbbRT network stack. Line 4 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 5). 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 9) 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

```
// 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<MacAddr> future_macaddr = ArpFind(local_dest);
  return future_macaddr.Then(
    // Lambda definition
    [buf = move(buf), proto](Future<EthAddr> f) {
      auto& eth_header = buf->Get<EthernetHeader>();
      eth_header.dst = f.Get();
      eth_header.src = Address();
      eth_header.type = htons(proto)
      Send(move(buf));
  });
}
```

Figure 2: Network code path to route and send and Ethernet frame.

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.

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 though we are not aware of any implementation for a native environment.

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.

3.6. Network Stack

One advantage of the EbbRT design is that application software can pass memory to and from hardware devices without copying. This is possible because all of physical memory is identity mapped and memory is never paged out. This means that aside from application managed regions of virtual memory, the address space is physically contiguous and pinned, a requirement for device DMA. Additionally, there is no address space separation between "system" components and "application" components. This allows EbbRT to avoid expensive data copies in cases where most systems must.

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.

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. 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 [40]. EbbRT stores connection state in an RCU [35] 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 have no cost. 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 flexibility our design enables by not pursuing complete compatibility with legacy interfaces. 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.

4. Evaluation

One primary goal of EbbRT is to enable the construction of high-performance library operating systems. To this end, we compare the performance of EbbRT library operating systems primarily with the Linux general purpose operating system. The other primary goal of EbbRT is to enable this degree of performance without inflicting a large maintenance burden on developers. We have not yet had the ability to evaluate this over a sufficient period of time. Instead, we evaluate EbbRT based on indicators of maintainability. This primarily involves describing code complexity, component reuse, and opportunities for offloading functionality.

Our performance evaluation compares the native EbbRT environment to Linux. We run our evaluations on a server containing two 12-core Xeon E5-2690 processors run at 2.6 GHz with 120 GB of RAM. For networked evaluations, we run client applications on another server containing a 20-core Xeon E5-2670 run at 2.5 GHz with 32 GB of RAM. Both servers contain a 10GbE Intel X520 network card (82599 chipset) directly connected to each other. Both machines have been configured to disable Turbo Boost and dynamic voltage frequency scaling. Additionally, we disable IRQ balancing and explicitly pin NIC IRQ affinity.

The EbbRT native library OS targets KVM guests. In order to evaluate its performance we run EbbRT applications within a virtual machine. All Linux tests run a minimal ramdisk image [3] of Debian 8.0 (jessie) with Linux kenel version 3.16. Unless otherwise stated, Linux applications are similarly run within a virtual machine. Virtual machines are deployed using QEMU verion 2.2.1 using the KVM kernel module with the `virtio-net` paravirtualized network card supported by the `vhost` kernel module. We enable multiqueue receive flow steering for multicore experiments. All Linux application threads are pinned to a dedicated physical core and we avoid using hyperthreads.

The evaluations are broken down into two parts; 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 two applications. While the hosted library is a critical component of our system 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 native environment.

4.1. Microbenchmarks

The first two micro-benchmarks evaluate the overheads of Ebb invocation and memory allocation. We conclude with a micro-benchmark that evaluates the latencies and bandwidth of our network stack. This benchmark exercises several of the system features including idle event processing, lambdas and `IOBuf` mechanisms.

Method	Cycles
Inline	1052
No Inline	4047
Virtual	5038
Inline Ebb	**1448**

Table 1: Object dispatch costs for 1000 invocations

4.1.1. Ebb Invocation Table 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 mem-

ory 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.

4.1.2. Memory Allocation Figure 3 shows per-core memory allocation throughput of EbbRT's default memory allocator as compared to the Linux glibc 2.19 and jemalloc[15] 3.6.0 allocators. Each core in parallel repeatedly measures the time to allocate and free an 8 B object ten times. We report the mean latency of one million measurements per-core. In this test, the Linux version of the memory allocation benchmark runs unvirtualized.

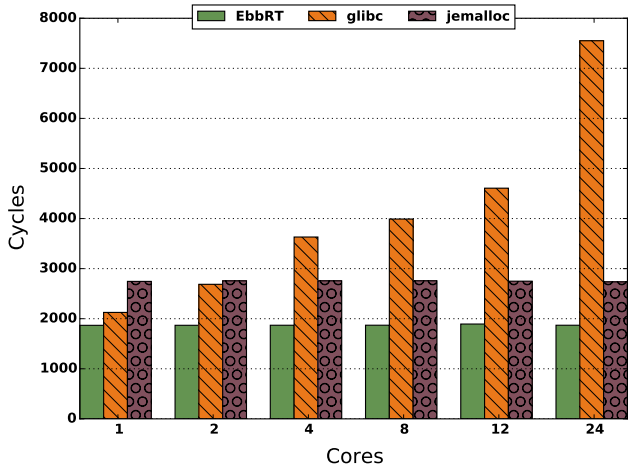

Figure 3: Memory allocation microbenchmark

The EbbRT allocator demonstrates linear scalability up to 24 cores. The glibc memory allocator exhibits poor scalability, with mean latency 3.8x that of EbbRT at 24 cores. Meanwhile, the jemalloc memory allocator also shows linear scalability but still 42% slower than the EbbRT allocator.

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

4.1.3. Network Stack In order to compare the performance of the EbbRT network stack to Linux's, we ported the NetPIPE [44] benchmark to EbbRT. NetPIPE is a popular ping-pong benchmark where the client sends a fixed-size message to the server which is echoed back after being completely received. With small message sizes, this benchmark illustrates the latency of sending and receiving data over TCP, whereas with large message sizes, the throughput of the path is stressed. In all cases, we run the same system on both ends.

Figure 4 shows the goodput achieved as a function of message size. Two EbbRT servers achieve a one-way latency of 9.7 µs for 64 B message sizes and are able to attain 4 Gbps of goodput with messages as small as 64 kB. In contrast, two Linux servers achieve a one-way latency of 15.9 µs for 64 B message sizes and are able to attain 4 Gbps of goodput with messages as small as 384 kB.

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 hardware to application and back enable significantly lower latency. 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 throughput until the network becomes the bottleneck.

Despite Linux having a highly optimized and mature network stack EbbRT achieves a 60% improvement in latency. The non-preemptive event-driven execution model which EbbRT provides is ideally suited for this class of applications. Additionally, the network stack implementation and low-level interface enable applications to attain high throughput by avoiding copies.

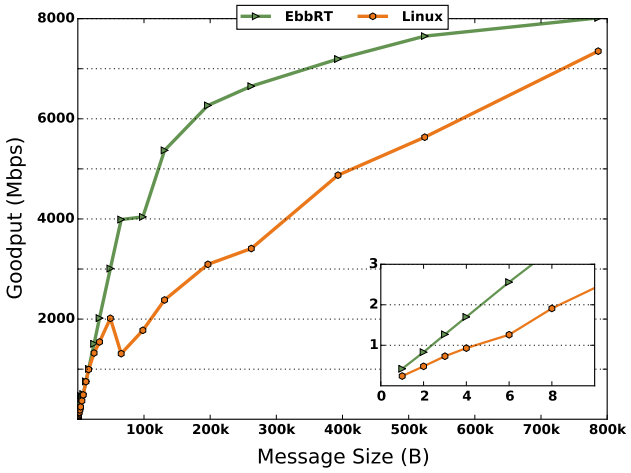

Figure 4: NetPIPE performance as a function of message size. Inset shows minimal message size region.

4.2. Memcached Performance

We evaluated memcached[17], an in memory key-value store. It has become a common benchmark in the examination and optimization of networked systems. Previous work has shown that memcached incurs significant OS overhead[26], 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 simple, multi-core application that supports the standard memcached binary protocol. Process management and logging is provided by a hosted process which then launches a native environment within which the performance critical network traffic handling occurs. Our implementation receives TCP data synchronously from the network card. It is then passed through the network stack and parsed in the application in order to construct a response, which is then sent out synchronously. Key-value pairs are stored in an RCU hash table to alleviate lock contention which is a common cause for poor scalability in memcached.

We compare our implementation of memcached to the standard implementation (version 1.4.22) running within both a Linux virtual machine as well as Linux running natively on our server. We run the `mutilate`[29] 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[7] 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 allowed to pipeline up to four requests per TCP connection.

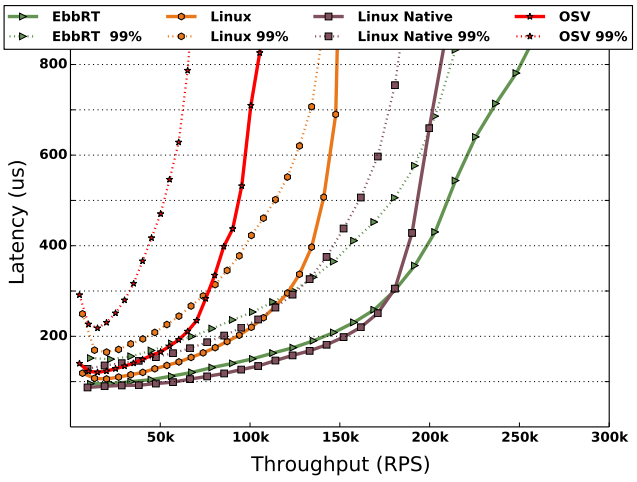

Figure 5: Memcached Single Core Performance

Our experimental infrastructure limits us to one machine to generate client load on the servers. This means that, with relatively few TCP connections, we are unable to induce significant queueing delays on the servers at very high throughput (when TCP retransmissions will throttle the load). Therefore, latency measurements near peak throughput may not be representative of a real workload.

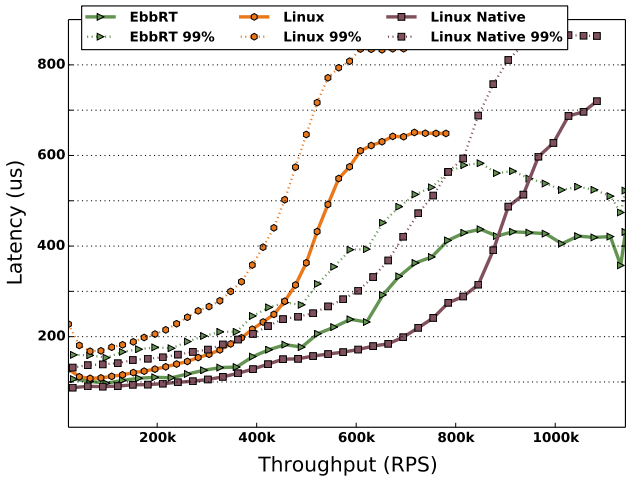

Figure 6: Memcached Multicore Performance

Figures 5 and 6 present the mean and 99th percentile latencies as a function of throughput for single core and four core servers. We present Linux running virtualized and natively (unvirtualized) in addition to EbbRT. At a 500 µs 99th percentile SLA, single core EbbRT is able to attain a 58% higher throughput than Linux within a VM and 11.7% higher than Linux native. At a 500 µs 99th percentile SLA, four core EbbRT is able to attain a 58% higher throughput than Linux within a VM and only 5% lower than Linux native. However, EbbRT is able to attain a higher peak throughput than Linux native. In fact, our 20-core client machine is unable to generate sufficient load to overwhelm the EbbRT server.

Additionally, we evaluated the performance of OS^v[27]. OS^v, like EbbRT, is a library operating system targeting cloud applications running in a virtualized environment. OS^v differs from EbbRT by providing a Linux ABI compatible environment for running a single application. Due to the similarities, it provides a natural comparison point to the EbbRT design. 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 four cores (omitted from the figure) due to a lack of multiqueue support in their `virtio-net` device driver. We contacted the system developers who were unable to discover an error in our configuration.

Significant effort has gone into improving the performance of memcached and similar key-value stores. This work includes client-side modifications [31], hardware acceleration [32, 25], use of UDP, and many others [35]. We believe that EbbRT would be a good target for all these optimizations, however we arbitrarily restricted ourselves to preserving memcached client compatibility over standard TCP. Even so, we were able to achieve significant performance improvements over an existing general purpose system.

One of our goals of EbbRT is to not hinder access to hardware interfaces. Our ability to implement memcached so that

it directly handles 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 significant performance advantages. The EbbRT primitives allowed us to achieve this with modest programmer effort. Many of the components and primitives such as `IOBufs` and RCU data structures were easily reused within the `memcached` application.

4.3. Node.js

We evaluated node.js, a popular Javascript environment for server-side applications. In comparison to memcached, node.js uses many more features of an operating system including virtual memory mapping, file and network I/O, periodic timers, etc. Often, systems can be demonstrated to show good performance for simple applications such as `memcached`, but as they grow to support more full-featured applications, their performance degrades. A key element of the EbbRT design is to provide an efficient base set of primitives on top of which individual applications can be constructed.

Node.js links with several 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 written in C++, and libuv[4], a library written in C which abstracts OS functionality and callback based event-driven execution. Porting V8 was relatively straightforward as EbbRT supports the C++ standard library which V8 depends on. Additional OS dependent functionality such as clocks, timers, and virtual memory are provided by the base Ebbs of the system.

Porting libuv required significantly more effort; there are over one hundred functions in the libuv interface which have OS specific implementations. We did not implement all of these functions, only those that were invoked in the process of running various Node.js applications.

The most complex aspect of the port is mapping the event loop to the underlying operating system. In EbbRT, the fundamental challenge was matching the stack conventions between EbbRT's event loop and libuv's expectation to have all events processed on a single stack. This involved constructing mechanisms for the necessary stack and register management. While this did not involve a significant amount of code, it was the majority of the intellectual effort. Our approach allows the libuv callbacks to be invoked directly from the hardware interrupt in the same way that the memcached application was able to.

One of the key results of the node.js port was the modest effort required to get it functional. And perhaps more importantly reusing the same software and mechanisms that we used to support memcached. This illustrates that EbbRT can support a broader class of event driven software beyond just hardware tuned network applications.

Finally, the port effort was significantly simplified by exploiting EbbRT's model of function offload. For example, filesystem access was implemented by invoking a *FileSystem* Ebb. Rather than implement a file system and hard disk driver within the EbbRT library OS, the Ebb offloaded calls to a representative running in a Linux process. Our implementation of the *FileSystem* Ebb is naïve, sending messages and incurring round trip costs for every access rather than caching data on local representatives. However, our simple approach allowed us to exploit functionality provided by Linux in order to accelerate the porting effort.

To make this concrete, we highlight that node.js and it's library dependencies total over one million lines of code (of which the majority is the v8 javascript engine). We wrote about 3000 lines of new code in order to support this large piece of software. This is critical to ensure that porting software to EbbRT is maintainable.

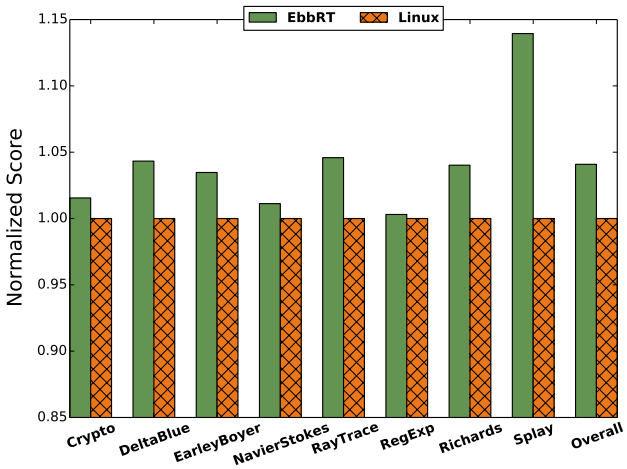

Figure 7: Nodejs Bench

To compare the performance of our system to that of Linux's, We launched node.js running version 7 of the V8 JavaScript benchmark suite. This is a collection of pure JavaScript benchmarks which perform no I/O and measure the performance of the V8 JavaScript engine. Figure 7 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. The total score is the geometric mean of the individual scores. For clarity, we normalize each score to that of Linux's.

EbbRT outperforms Linux on all benchmarks with a 4.09% improvement in total score. In particular, EbbRT is able to attain a 13.9% improvement in the memory intensive `Splay` benchmark. While we made no modifications to the V8 implementation, EbbRT is still able to achieve a noticeable improvement. We attribute this to better virtual memory management and the lack of scheduling interrupts. EbbRT aggressively maps in memory allocated by V8 and therefore suffers no page faults. Additionally our non-preemptive execution environment prevents unnecessary timer interrupts and cache pollution due to OS execution. Lastly, we evaluated a node.js webserver. The webserver uses the builtin `http` module and

	Mean	99th Percentile
EbbRT	90.54 µs	123.00 µs
Linux	112.83 µs	199.00 µs

Table 2: Node.js Webserver Latency

responds to each `GET` request with a small static response, totaling 148 bytes. We use the `wrk`[21] benchmark to place moderate load on the server and measure mean and 99th percentile latencies. The results of running this benchmark can be seen in Table 2. We see that the webserver running on top of Linux has a 24.61% higher mean latency than the same webserver on top of EbbRT. Linux's 99th percentile latency is 61.78% higher than EbbRT's.

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 performance. This allows unmodified node.js applications to directly gain performance advantages by running on top of EbbRT. We believe that our system also enables future optimizations which we have not yet been explored. For example, one could modify V8 to use direct access to page tables to improve garbage collection. While we expect greater performance can be achieved via customization of the application, it is a significant result that we were able to support such a large application within the same light-weight, high-performance environment that was used for memcached. This encourages reuse of various optimizations within a shared environment.

5. Related Work

While our work combines and builds on many aspects of prior systems research, we are particularly influenced by results from work in: hybrid structure, customizable systems, kernel toolkits, and OS bypass.

Our hybrid structure was influenced by CNK[36], Libra[6], and Azul[45]. All three pursued a distributed approach where a customized library OS is used in conjunction with a general purpose OS. In the case of Libra and Azul a library OS was constructed to support a JVM and in the case of CNK a Linux MPI process. All three systems exploited a process on the general purpose system to acts as a proxy for the library OS instance. System calls and services not implemented by the library OS are forwarded to the proxy via network exchanges. From the perspective of the general purpose OS the proxy allows the library OS instance to appear like a local application process which is managed like other local processes. EbbRT generalizes and advances this model. EbbRT give a developer the ability to generate an application specific hybrid structure. Ebbs allow a developer to off load function in a fine grain fashion and in both directions.

There has been a broad body of work that has explored system support for customization. Spin[9], Exokernel[14], Vino[43], Synthesis[39], Cache Kernel[11], and more generally microkernels such as L4[30], all explored structuring and primitives for composing and or tailoring system functionality for applications. By taking a distributed library OS approach in which a node is dedicated to one application function EbbRT can take a more extreme approach to customization. Having no protection or multiplexing concerns application code can be hoisted all the way down into hardware specific paths making that instance of EbbRT customized in the extreme to the application. EbbRT does not just blur the line between application and systems code it enables its selective removal.

EbbRT also relies heavily on a toolkit model to compose and specialize system functionality. In EbbRT, Ebbs are the fundamental unit of encapsulating functionality. Our goal, like that of the Flux OSKit[18], is to have developers construct reusable components. We are, however, equally interested in developers encapsulating highly specialized and performant function that need not conform to a broader ontology of component. From this perspective Ebbs shares similarity to other systems that have used components to encapsulate distributed structures such as Globe[46], KTK[19], and K42[28] which not only allow for composition, but also encapsulate communications and performance tradeoffs.

More recently library OSes such as Mirage[33] and OSv[27] explore two different models for exposing customized system function to applications. The former exploits a language level managed runtime interface and the later exposes the Linux ABI. In contrast, EbbRT provides a low level C++ runtime and associated toolchain which allows native code to be compiled directly to our system. A library or application developer can freely mix and match new and existing code when developing to EbbRT. As demonstrated by our port of the Google V8 Javascript engine, the base EbbRT software can be used to construct managed runtimes. Our goal is to allow developers to explore various constructions that meet their needs for compatibility while not precluding extreme customization.

The EbbRT work is not the first to observe that there is an opportunity to improve performance by by-passing kernel interfaces and protection. Most recently, and perhaps most similar to EbbRT with respect to targeting datacenter scale systems, Arrakis[37] and IX[8] both advocate for exploiting hardware virtualization to allow applications directly access devices for their data path while maintaining protected mechanisms for establishing this access. EbbRT is able to provide comparable performance in addition to a set of primitives and software structure to enable development of new per-application optimizations in a maintainable fashion.

6. Conclusion

We have presented EbbRT, a framework for constructing customizable library operating systems without the typical maintenance costs of custom systems. Our evaluation demonstrates some of the performance advantages that can be achieved with our design. Our design enables developers to structure software as reusable components. Our hybrid architecture enables rapid application development by offloading non-performance critical functionality.

References

[1] Boost C++. http://www.boost.org/.

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

[3] debirf. http://cmrg.fifthhorseman.net/wiki/debirf.

[4] libuv. http://libuv.org.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[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*, page 14. USENIX Association, 1994.

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

[13] 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.

[14] 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.

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

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

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

[18] 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.

[19] Ahmed Gheith, Bodhisattwa Mukherjee, Dilma Silva, and Karsten Schwan. KTK: Kernel support for configurable objects and invocations. In *Configurable Distributed Systems, 1994., Proceedings of 2nd International Workshop on*, pages 92–103. IEEE, 1994.

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

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

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

[23] Google. V8 javascript engine.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] Jochen Liedtke. Improving IPC by Kernel Design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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].

[36] 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.

[37] 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, Broomfield, CO, October 2014. USENIX Association.

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

[39] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis Kernel. *Computing Systems*, 1(1):11–32, 1988.

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

[41] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference*, pages 101–112, 2012.

[42] 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.

[43] Christopher Small and Margo Seltzer. Vino: An Integrated Platform for Operating Systems and Database Research. Technical report, Citeseer, 1994.

[44] 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.

[45] 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, New York, NY, USA, 2011. ACM.

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

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

[48] David A Wheeler. generated using david a. wheeler's 'sloccount'.

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