EbbRT: Elastic Building Block Runtime - Case Studies

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

Abstract

We present a new systems runtime, EbbRT, for cloud hosted applications. EbbRT takes a different approach to the role operating systems play in cloud computing. It supports stitching application functionality across nodes running commodity OSs and nodes running specialized application specific software that only execute what is necessary to accelerate core functions of the application. In doing so, it allows tradeoffs between efficiency, developer productivity, and exploitation of elasticity and scale.

EbbRT, as a software model, is a framework for constructing applications as collections of standard application software and Elastic Building Blocks (Ebbs). Elastic Building Blocks are components that encapsulate runtime software objects and are implemented to exploit the raw access, scale and elasticity of IaaS resources to accelerate critical application functionality. This paper presents the EbbRT architecture, our prototype and experimental evaluation of the prototype under three different application scenarios.

1 Introduction

EbbRT is a new distributed application runtime for Infrastructure as a Service (IaaS) clouds. It is designed to allow applications to efficiently and elastically exploit large numbers of virtual or physical nodes. It is also designed to reduce the development and operational costs of distributed cloud applications.

EbbRT is designed around three key methodologies. First, it is a realization of the *MultiLibOS* model[**?**], where applications are distributed across a mix of general purpose OSs and specialized library OSs. This model takes advantage of library OSs by using them as accelerators to one or more processes running on a commodity OS. The small footprint of system software provided by the library OS permits the application to rapidly bootstrap and exploit new nodes. The library can be heavily customized to adapt the use of the hardware to the specific application functionality being deployed to it.

Second, it adopts a lightweight, non-preemptive event-driven programming model. This model is well matched to the communication and protocol processing nature of many cloud applications. The implementation in a library OS eliminates traditional OS boundaries and allows the mixing of application logic and device interaction. This enables applications to customize the processing of any hardware interrupt to improve processor efficiency for handling external interactions — a critical function of these applications.

Third it adopts a partitioned object model called *Elastic Building Blocks* (Ebbs). A partitioned object model uses object orientation both as a methodology for structuring the software as components and as a way to encapsulate and reuse distributed and multi-core data structures and operations. Ebbs provides EbbRT with two fundamental runtime aspects: 1) all system functionality in the library OS is provided by Ebbs, allowing a developer to compose, customize or configure the library OS to meet an application's specialized requirement, 2) the Ebb component infrastructure provides developers with a programming model for expressing elasticity and managing the complexity of distributed and multi-core applications.

This paper describes the EbbRT architecture, prototype and three use cases that explore different aspects of the design. These use cases are: 1) an EbbRT implementation of memcached[10], 2) a port of the V8[12] javascript engine and node.js[14], and 3) the integration of an elastically distributed matrix object into the Sage Math[1] environment.

The goals of EbbRT are to enable the efficiency advantages associated with specialized system software while supporting the high development productivity associated with reusable general purpose software. The evaluation provides initial evidence of the suitability of EbbRT to its goals.

Efficiency: All three use cases demonstrate improvement in efficiency over the application running on linux. The third example also demonstrates that we can boot and initialize new nodes rapidly to efficiently map the elasticity of the application onto the elasticity afforded by cloud resources.

Productivity: Each of the use cases demonstrate different aspects of productivity. The first suggests that we can write a very simple event-driven application without all the complexity needed to achieve high performance on a general purpose OS. The second demonstrates that we can support a complex, managed run time, allowing existing applications of the runtime to execute without modification. The third demonstrates that we can modify complex applications incrementally to exploit elasticity in novel ways.

While new application specific objects were introduced and different existing objects were customized in all three use cases, the core EbbRT framework remained stable and sufficient; suggesting that it may be of broad applicability.

The rest of the paper is structured as follows: The motivation for the design of EbbRT is presented in Section 2. Section 3 describes how the three key methodologies in EbbRT relates to previous work. Section 4 provides an overview of EbbRT's architecture and describe our prototype. Section 5 evaluates the prototype using the three use cases.

2 Motivation

Efficiency for applications using a cloud is determined by; the instructions per cycle (IPC) on the nodes used , the overheads to perform I/O for these communication bound applications, and the ability to rapidly grow and shrink resources to match the application's varying demands.

Today, distributed cloud applications rely on middleware that run on top of legacy operating systems. We believe that much greater efficiency will be achieved if the support for complex cloud applications is done at the operating system level. We believe that changes in our model of an OS will be critical if we want to rapidly grow and shrink resources, exploit very low latency communication between nodes, and enable complex heterogeneous systems. Unfortunately, it is a difficult and complex task to re-write our existing operating system to explore different OS functionality; hence innovation is largely happening above the OS.

The complexity of our existing operating systems is driven by the need to concurrently handle multiple tenants and applications with diverse functional and resource management requirements. However, in an IaaS cloud, nodes are typically dedicated to a single tenant and distributed application; reducing much of the security and resource management burden. Further, distributed applications typically adopt a service-oriented architecture, where sets of nodes are dedicated to specific application functionality (e.g. database, in memory cache, application logic service, etc.). Hence, we can see that much of the complexity of our legacy OSs is not needed for many of the nodes in an IaaS cloud.

EbbRT takes advantage of the simplified per-node demands in an IaaS cloud to exploit a different OS model; the MultiLibOS model. In this simpler environment we have the opportunity to explore new functionality to efficiently meet the needs of cloud applications; namely, direct OS support for the event model common to these applications, and a programming model (Ebbs) to help developers reason about complex distributed and multi-core code.

3 Related Work

EbbRT draws from work on library OSs, event driven software, and the use of partitioned object models in both multiprocessor and distributed systems. In this section we provide the context for each and how EbbRT relates to and builds on this previous work.

3.1 Library OSes

Library operating systems[9] organize a single application and the OS functionality it requires into a single address space and protection domain. The application code links directly to the OS code and invokes it via a standard function call. Library OSs enable reductions in overheads and the opportunity to specialize and tailor system functionality and interfaces for a particular application's needs.

In recent years, several efforts have explored how virtualization can be leveraged to provide benefits by directly executing applications in their own VMs linked with a library OS [20, 24, 16]. These benefits range from improved security to higher performance. The basic approach is to extract out a particular function of an application and run it along with a library OS in its own virtual machine.

Generally, library OSs provide some level of ABI[24, 6] or API[20, 16, 19, 23, 3] compatibility. This has been done in three ways; 1) supporting C and C++ standard libraries, 2) porting of managed language runtimes such as Java[3, 16] and Ocaml[20] and/or 3) using a shim layer to forward system calls to an instance of a standard OS running in a different VM.

EbbRT: EbbRT provides a distributed runtime which allows processes of general purpose systems to launch *back-end* nodes running a lightweight library OS. The runtime allows for function offloading from the library OS to the general purpose system and vice-versa. From a users perspective, an EbbRT application appears like any

other process that is launched, owned and managed by the user. This *front-end* process serves both as the user's access point to the application, and also as the access point for the back-end nodes to the front-end OS resources such as files and external I/O channels. There are cases under which an EbbRT application might exploit more than one front-end node to reduce contention and improve fault tolerance. Our current work, however, focuses on the case of an application having a single front-end process.

EbbRT exploits library OSs for the back-ends to enable application and hardware specific optimizations. In particular, the event and Ebb primitives described in the next sections can interact with the hardware at a very low level. All services implemented in the library OS can be tuned to the specific needs of the application.

The EbbRT library OS is distributed with a port of the C and C++ standard libraries. OS functionality is provided to these libraries by invoking methods of EbbRT components. These components can be implemented to communicate with the front-end to alleviate the burden for native local implementation where appropriate. This approach to compatibility is tractable for supporting managed runtime environments as demonstrated by our port of the node.js runtime.

Other library OSs have been developed to be deployed on a cloud [3, 20], but EbbRT is the first distributed library OS we are aware of. Other research groups exploring new operating systems for the cloud [25, 29] are not focused on a library OS model. We believe that the asymmetric model adopted by EbbRT, that includes both general purpose and library OSs, is both unique and critical to allowing us to aggressively explore new technologies while supporting real applications.

3.2 Event Driven Software

Event driven architectures and associated programming models are designed to reflect and enable applications that must respond to asynchronous actions. Typically, this is done with a callback model such that when an action occurs, a programmer-specified routine is invoked by the system in response.

Hardware inherently supports an event driven model through its interrupt and exception support. As such, the software at the lowest level of most operating systems is written in an event driven manner directly on top of the hardware mechanisms. Operating system research has also explored how systems software can be better structured to directly support network based application processing which is inherently event driven[28, 17, 22].

The suitability of event driven programming to network application programming has made it popular for cloud and internet applications, so much so that the legacy process and thread models of commodity OSs are often abandoned in favor of lighter weight user-level primitives for supporting event driven programming via some form of explicit stack switching. Similarly, many user-level libraries such as Boost.ASIO and libuv have been developed to ease the burden of writing portable event driven applications on top of commodity OS features. Further, web application runtimes and languages have widely embraced event driven models and incorporated features such as promises and lambdas to better facilitate the use of continuations that are often required when programming in an event driven fashion.

EbbRT: EbbRT supports a non-preemptive event driven execution model. Not only does this match the trends of IaaS application programming, but also, it allows for a lightweight implementation which maps directly to hardware mechanisms. Hardware interrupts cause application event handlers to be invoked. Event handlers run to completion with interrupts disabled. This allows application software to execute directly off of hardware interrupts without the need for thread scheduling and context switches. In order to support blocking programming interfaces, software can voluntarily yield the processor, saving its state, in order to dispatch further events. In contrast to other event driven operating systems [27, 2], where continuations required by an event driven system added tremendous programmer complexity, we make extensive use of C++11 language features, such as lambdas, to reduce programmer complexity.

This execution model allows for a number of optimizations. Per-core data structures can be reused across many events without the need to synchronize due to the lack of pre-emption. Additionally, because interrupts are only enabled at the termination of an event-handler, state does not need to be saved when an interrupt occurs. At a larger level, much of the complexity of a scheduling infrastructure is avoided, allowing applications to easily control event execution.

The lack of preemption means that a long running event will make the system non-responsive to new events. Software developed directly to the base event model needs to be carefully designed to avoid this. We have so far found it natural to implement the core system software under this constraint. In scenarios where a more complex execution model is required, the event infrastructure can serve as a natural foundation for threads and future schedulers.

While much previous work has studied event driven models and non-preemptive threading, EbbRT's novelty lies in exposing the combination of these techniques to

general purpose cloud applications in a library OS model. As we will see, direct low-level support of events in a library OS can result in major effiency gains for event-driven cloud applications.

3.3 Partitioned Object Models

The development of high-performance, parallel software is non-trivial. The concurrency and locality management needed for good performance can add considerable complexity. Prior work has demonstrated that a *partitioned object* model can facilitate the construction of parallel system software, both for distributed and shared memory systems. In a partitioned object model, an object is internally composed of a set of distributed *representatives*. Each representative locally services requests, possibly collaborating with one or more other representatives of the same partitioned object instance. Cooperatively, all the representatives of the partitioned object implement the complete functionality of the object. To the clients of a partitioned object, the object appears and behaves like a traditional object.

The distributed nature of partitioned object models make them ideally suited for the design of both multiprocessor and distributed system software, which often requires a high degree of modularity and yet benefits from the sharing, replicating and partitioning of data on a per-object basis. Fragmented Objects(FOs) [7, 21, 26] and Distributed Shared Objects(DSOs) [5, 13] both explore the use of a partitioned object model as a programming abstraction for coping with the latencies in a distributed network environment, LAN and WAN respectively. Clustered Objects[11, 18, 4] demonstrated the effectiveness of a partitioned object model in the construction of multiprocessor operating systems.

EbbRT: Core to EbbRT is a partitioned object model called *Elastic Building Blocks* (Ebbs) that provide a model for software components to independently and elastically expand to react to system-wide demand. An Ebb is associated with its EbbId, a system wide unique identifier. When a client invokes an interface of an Ebb, the request is directed to a per-core representative which may communicate with other representatives on other cores or nodes within the system to fulfill the request. EbbRT puts no restrictions on how the representatives of an object must communicate or organize themselves; allowing Ebbs to be used for a wide range of different software components.

Object invocations are directed efficiently to a representative by exploiting a virtual memory region backed by different physical pages on each core. Representatives are created on demand. When a request is made to a non-existent representative a programmer specified *fault handler* is invoked in order to construct it.

While EbbRT draws heavily from previous work on partitioned objects, it differs by providing a unified framework for reasoning about both distributed and multi-core parallelism. In addition, the combination in an event-driven library OS allows the application developer to directly reason about parallelism resulting from interactions with the hardware.

4 Architecture and Prototype

In this section we present the architecture of EbbRT and our prototype of it.

4.1 Architecture

As discussed, EbbRT is structured as a MultiLibOS and supports a single instance of an application distributed across a set of IaaS provided nodes. Figure 1 illustrates the three layers of the EbbRT software architecture. The

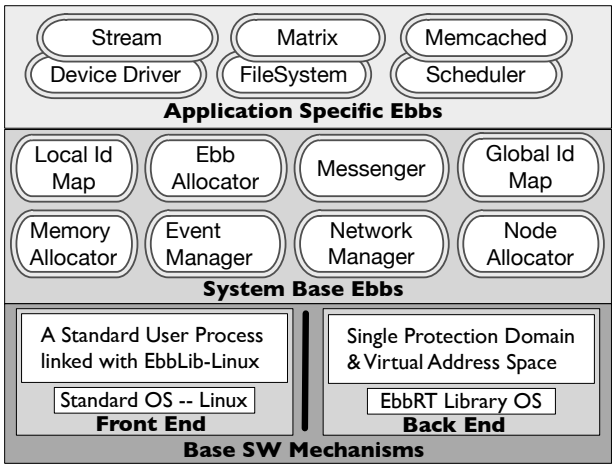

Figure 1: EbbRT Architecture

lowest layer provides the base software mechanisms for constructing the address spaces that the Ebb application will run in. The rest of the EbbRT software, illustrated in the top two layers, takes the form of Ebbs and execution occurs on lightweight non-preemptive events. The system base Ebb layer is mandatory and provides the support for Ebbs, events and off-node communication. While the Ebbs in this layer are mandatory, the implementations themselves can be customized for any particular application's needs or hardware features. Collectively, these Ebbs define the base interfaces and function of EbbRT, everything else is specific to an application and linked in

as necessary. While implementations may differ, the same interfaces are provided on the front-ends and back-ends.

4.1.1 Base Software mechanisms

In the case of a front-end, a standard user process linked to the EbbRT library serves as the EbbRT address space on the node. However, the back-ends use custom boot images that contain the base EbbRT Library OS software that bootstraps the node and maps all the nodes resources to a single virtual address space running with ring zero privilege. All physical memory is identity mapped into this address space.

The base software mechanisms include interfaces for establishing flexible virtual mappings. For example, this includes virtual memory regions for applications to specify their own page fault handler. Additionally, debugging facilities and boot code are distributed as part of this layer. A more complete discussion of the base software mechanisms is out of scope for this paper

4.1.2 System Base Ebbs

The system base Ebbs provide interfaces that additional Ebbs can be built on. A single instance of each of these Ebbs is provided when the application is initialized. They are fully replicated and their internal representative construction happens on demand as nodes are added to the application and the instances themselves are accessed on a particular node. This layer provides mechanisms for dynamic allocation and configuration of Ebbs as well as communication between them. We defer further description of these Ebbs to our discussion of them in the three use cases.

4.1.3 Application Specific Ebbs

Above the base Ebb layer, arbitrary application specific Ebbs can be constructed. A critical goal of the architecture is to permit a high degree of specialization and customization for an application's needs through composition and configuration.

EbbRT's component architecture was chosen to make it viable to construct reusable libraries of Ebbs. Fine-grained, object-oriented decomposition provides many degrees of freedom to customize even its most basic system functionality, such as the event processing loop and interrupt dispatch. In this case, it could be accomplished with an application customized implementation of the *Event Manager*. EbbRT expects many traditional bundled-in features of a library OS to be provided as independent libraries of Ebbs. These include additional device support, files, network protocols and other system abstractions. The Ebb library organization extends into the application as well. The support for libraries of application Ebbs that provide application specific components such as scalable and elastic matrices, are a core value of the architecture. As such, the final runtime structure of an Ebb application should be a composition of Ebb instances that are solely focused and necessary for the application specific processing that is to be done.

4.2 Prototype

Our EbbRT prototype consists of a main body of C++ software from which two libraries are generated. The source is composed of approximately 9600 lines of C++ and 330 lines of assembly code. One library generated is a standard Linux library. This front-end library can be linked either statically or dynamically to a Linux application. The other library is a x86-64 library that can be used to create a boot image that contains the EbbRT library OS and can be launched in a KVM virtual machine. All software targeting the EbbRT library OS is built using a port of the GNU C++ toolchain. This allows EbbRT applications to make use of the C and C++ standard libraries and runtime features such as exceptions.

In order to explore EbbRT, we have constructed a simple synthetic IaaS that launches virtual machines. Using our IaaS interface, a user can dynamically acquire nodes and boot them with arbitrary images. All nodes of a particular user are placed on a user specific private virtual network. EbbRT applications can allocate additional nodes by invoking a system provided Ebb that calls out to our IaaS daemon. This daemon launches virtual machines to boot with the specified image and set of arguments. Another implementation of this Ebb has been developed targeting the HP Public Cloud.

While our prototype, as stated above, is realized on KVM, nothing precludes EbbRT from running on a physical host. We expect that as IaaS providers evolve to hardware systems that make physical provisioning viable, our prototype can be modified to provide even greater value.

5 Evaluation

We evaluate and explore the EbbRT prototype through three case studies that evaluate and demonstrate different aspects of the system. The first use case, memcached, demonstrates the performance potential possible with our approach. The second, node.js, discusses our experience porting a managed runtime and demonstrates the viability of supporting unmodified applications. The third, Sage,

demonstrates the value of the asymmetric model, allowing software packages to be incrementally modified to exploit the elasticity and scale of IaaS environments.

5.1 Memcached

This case study describes a memcached[10] server, implemented with EbbRT, to produce a bootable image. This use case demonstrates that EbbRT enables very simple application code to fully exploit hardware and illustrates the use of the event driven execution model for supporting a cloud application.

Memcached implements a simple key-value store. It is designed to be highly performant, and has become a common benchmark in the examination and optimization of networked systems. Previous work has shown that memcached incurs significant OS overhead [15], and hence is a natural target for a library OS.

5.1.1 Implementation

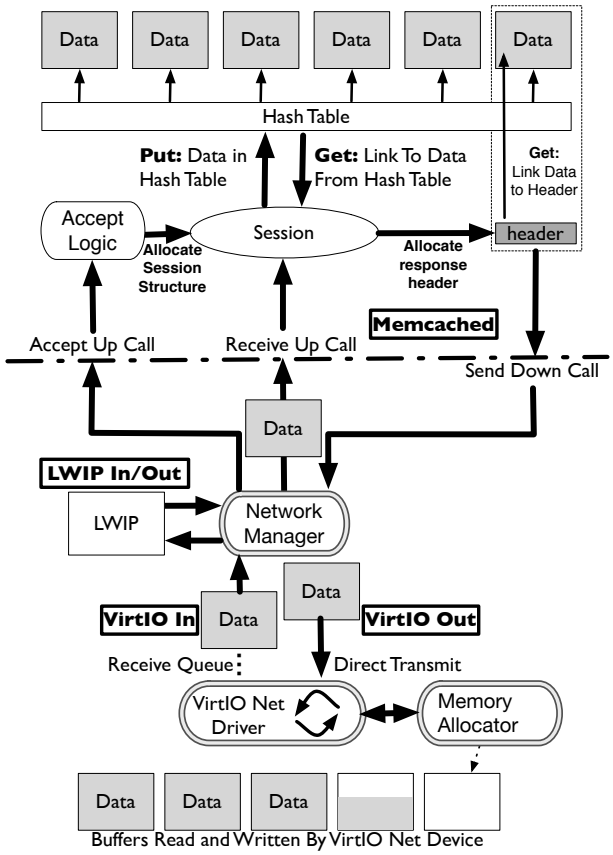

Figure 2: EbbRT Memcached Application

The back-end EbbRT memcached server is a simple single-core application that supports the standard memcached binary protocol. Our implementation is only 277 lines of original C++ code. To a developer with knowledge of the EbbRT interfaces, this basic application can be developed in a single afternoon.

The upper portion of Figure 2, above the dashed line, illustrates the logic of our application and its primary data structures. The application is constructed around two events, *Accept* and *Receive*, denoted by the two up arrows entering the memcached portion. These are registered with and invoked by the *Network Manager*. The application code also invokes the *Network Manager* to send responses (illustrated with the downward arrow exiting the memcached portion). Given that only a single core is used by the application and EbbRT's non-preemptive event model, all the call backs are executed sequentially and run to completion.

The memcached code registers a function to be invoked when a new connection is accepted. For each connection the memcached logic creates a session object to process requests on the connection. The application registers to receive up calls when data is received on the associated connection.

The lower portion of Figure 2 illustrates how the EbbRT library OS internals interact with the memcached application. At the bottom the Network Interface Card (The VirtIO Net paravirtualized device) deposits Ethernet frames into memory buffers. When a buffer is written to, the device marks an associated descriptor as dirty. When all buffers are used, the device will drop new Ethernet frames received.

In the steady higher load states the EbbRT network device driver Ebb (shown at the bottom of the diagram) uses a re-occurring *idle* event which runs when no other events exist. This causes the device state to be polled when data is available rather than repeatedly taking interrupts. If no used buffers are found, interrupts are enabled on the device and the idle event handler is unregistered. This allows the processor to drop into a low power state if no requests are being received. If a used buffer is found, a descriptor to the buffer is passed to the *Network Manager* for further processing. The memory containing the payload is never copied, a descriptor is passed through the networking stack all the way to the application.

The *Network Manager* wraps the Light Weight IP (lwIP) [8] networking stack that is linked and ported to the EbbRT Library OS. This software processes the frame and identifies it with a TCP connection. The Network Manager then invokes the application registered callback for data reception on that connection.

The memcached application logic will then run to com-

pletion (including any network sends) and return back to this point. This code will continue to return back until the top frame of the event is exited. At this point, the Event Manager's event loop logic will briefly enable interrupts to process any pending interrupts. In the memcached scenario, the only interrupts that might occur are timer interrupts associated with network processing (e.g. retransmits, delayed ACKs). The event loop will disable interrupts and then execute the idle handler.

Jointly the upper and lower portions of Figure 2 illustrates the entire path of input processing. What's critical to note is how packet data and memory moves up from the NIC to the application on a single event with no preemption and no memory copies. This creates a run to completion packet processing model that encompasses all software logic, device, protocol stack and application. In fact, the application hash table directly stores buffers that were originally allocated by the device driver when a *set* is invoked. When *get* is invoked this same memory can be chained along with a newly allocated message header for direct transmission by the device.

5.1.2 Evaluation

Environment Experimental measurements were gathered on a single Dell PowerEdge R620 server, equipped with two 10-core Intel Xeon EV-2670v2 processors, the Intel C6202 chipset, and 32GB of DDR3 RAM. The host system ran CentOS 6.5 with Linux 2.6.32. Guest Linux VMs ran Debian 7.4 with Linux 3.2.0. Our IaaS simulation daemon, that ran on the host, was configured to deploy qemu-kvm (version 1.7.5) instances. Each KVM guest (EbbRT or Linux) was each given a single VCPU, pinned explicitly to an inactive physical core, and 4GB of memory. The guests were connected to the physical network via an Ethernet bridge on the host machine, and used KVM vhost-net.

To evaluate the performance of our memcached implementation we ran the memaslap benchmark included with memcached. Memaslap is run on a remote machine connected to the host machine via a switch and a gigabit Ethernet link. In this way, each test accounts for the round trip latencies of a single network hop (0.10 ms). Memaslap is configured to do a 9:1 ratio of *get* operations to *set* operations. We run the same experiments on the standard Linux memcached implementation to provide a comparison.

Figure 3 shows the throughput of memcached for a small payload as the number of concurrent requests from the client increase. Furthermore, the EbbRT implementation's throughput peaks at around 64 connections, with about 1.7 times the throughput of the Linux implementation with the same concurrency.

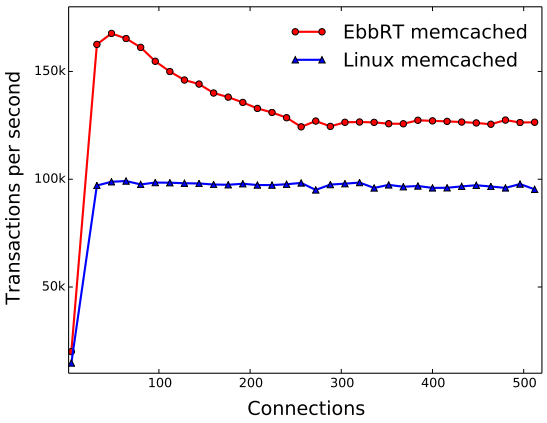

Figure 3: Memcached throughput as a function of number of concurrent clients

Figure 3 also illustrates that with more than 64 connections, Linux's throughput maintains the same rate, while EbbRT's gradually degrades. The source of this degradation in EbbRT is shown in Figure 4. The main degradation is in the LWIP receive performance. Upon examination of the code, we discovered that each receive ends up traversing a linked list of all the connections. Moreover, LWIP moves the most recently used connection to the front of the list, causing most accesses for this benchmark to need to traverse the entire list. We believe that this degradation stops once concurrency reaches 256 because the number of concurrent packets exceeds the ring buffer of the device, resulting in TCP retries, and slowing down some clients. This results in the LWIP optimization having less of a negative consequence.

Figure 4 also demonstrates another important point. Even in EbbRT's highly optimized environment, at low concurrency, the total time spent in application code is only around 15% of the total execution time for a single memcached operation. This demonstrates the importance of optimizing system software for this kind of application.

Figure 5 shows the performance of memcached, with a fixed concurrency of 64 sockets (EbbRT's peak) as the payload size increases. EbbRT is able to process packets at a high enough rate to saturate the network at around 800 bytes, while the implementation on Linux is not able to saturate the network until packets are around 2 kilobytes. Note, the dip in performance for both implementations occurs when they segment packets across multiple ethernet frames. We are investigating the source of the sawtooth shown in EbbRT's throughput as payload sizes become large.

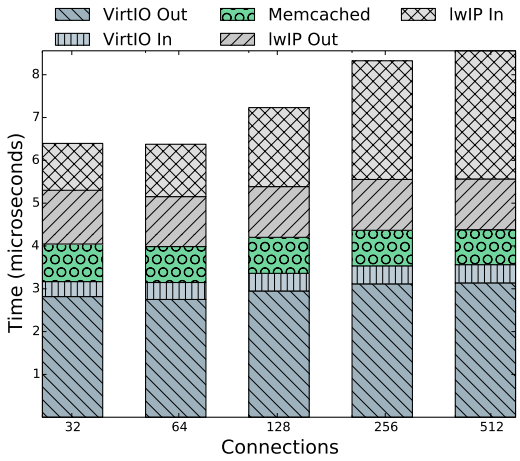

Figure 4: Per-request latency breakdown of EbbRT memcached

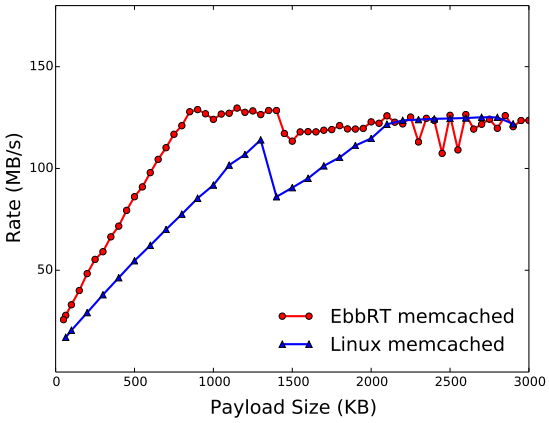

Figure 5: Memcached throughput as a function of payload size

Discussion The performance data above indicates that the EbbRT implementation of memcached is able to achieve major performance gains over the Linux based implementation. Under peak conditions, with small payloads the EbbRT memcached implementation is able to handle 1.7 times as many requests as the Linux implementation and saturate the network at a much smaller packet size. The current performance degradation is due to the LWIP library used for TCP/IP. While LWIP is small and simple to port it is not designed or optimized for high-performance compared to Linux's mature and server grade protocol stack. Not only is its performance under load problematic, but it does not have support for hardware optimizations like segmentation offload. A more performant TCP implementation for EbbRT is currently under way.

To understand why performance is so much better with the EbbRT implementation, its worthwhile to compare what has to happen with the Linux implementation to the EbbRT based one. With Linux, the application calls epoll (a context switch), the kernel wakes it up when a packet arrives (context switch), the application then reads the data (context switch and copy) and then writes a reply (another context switch and copy). In contrast with EbbRT all these system calls and copies are avoided; EbbRT results in fewer context switches and buffer copies than Linux on every client request.

One option with an EbbRT implementation is to make optimizations that are application specific and perform poorly for other applications. For example, we found that under extremely heavy load a minor performance improvement resulted when we handled packets in reverse order, since starving some clients under very heavy load limited the number of TCP timeouts observed. This same change resulted in orders of magnitude degradation on a simple TCP streaming benchmark. While this optimization is not that significant (and was not used while gathering the earlier results), it demonstrates how a brittle change that only performs well for one application is a reasonable option in a system like EbbRT.

The results obtained are consistent with Chronos [15], which achieved similar performance on top of Linux by bypassing the operating system. The two projects have adopted very different approaches to achieve the same goal. It is certainly possible to provide functions on a general purpose OS that allow applications to be developed that bypass any specific OS functionality. However, doing so results in significant OS complexity. Now that one can easily provision nodes on an IaaS cloud, we believe the EbbRT design provides a natural alternative to supporting performance demanding applications that are a poor match for general purpose systems.

Perhaps the most important result of this memcached experiment is that the application took only 277 lines of original code to implement. The effort in designing EbbRT has made it possible for very simple applications to be written very close to the hardware.

Our experience in developing memcached also demonstrates that EbbRT is a natural match for the intrinsic event driven nature of this kind of application. These applications, which are fundamentally about handling networking events, are a mismatch for general purpose OSes, on which an event model is constructed on top of threads within a protection domain isolated from the device.

It should be noted that our current implementation of memcached is limited to one core. While EbbRT is designed to support multi-core applications, the LWIP library limits our multicore performance. Removing this barrier is, for now, future work.

5.2 NodeJS

This case study describes the port of node.js, a JavaScript environment for server-side applications, to EbbRT. It illustrates three points: 1) EbbRT can support complex managed code environments, allowing existing software to run unmodified on the library OS back ends with performance advantages as compared to Linux. 2) EbbRT's non-preemptive, event-driven execution environment is suitable for large, complex applications such as node.js. 3) OS functionality can be offloaded to a general purpose OS, easing the effort of porting to the EbbRT library OS.

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, Google's JavaScript engine written in C++, and libuv, 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 from the hardware interrupt in the same way that the memcached application was able to.

In particular, the networking interfaces provided by libuv were translated into invocations of the Network Manager Ebb. Callbacks were installed such that they were invoked on a single stack. This was sufficient to allow us to run node.js applications including TCP stream processors and web servers.

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.

5.2.1 Evaluation

Memcached allowed us to explore communication oriented software. Largely, optimizations of the protocol stack would benefit any networked application running on node.js. Instead, our evaluation focuses on exploring how the port of node.js enables runtime managed code to execute on the EbbRT library OS. To do this we study a standard JavaScript benchmark. In particular, we deployed the EbbRT library OS without a corresponding front-end to isolate the library OS's support for the managed runtime. The standalone library OS launches node.js with version 7 of the V8 JavaScript benchmark suite. This was compared to a Linux appliance booted with the Ubuntu 14.04 server kernel (3.13.0-12-generic) and a custom RAM disk which starts node.js with the V8 benchmark suite via the init script. Both systems power off once the benchmark suite terminates and node.js exits. All experiments were performed on the same server as the memcached experiments.

The V8 JavaScript benchmark suite runs seven individual benchmarks exercising different aspects of the engine such as memory management, code generation, and various compute intensive operations. It is important to note that none of these benchmarks perform any I/O. The suite reports a score for each benchmark. The score is 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.

Figure 6 illustrates the arithmetic mean of all reported benchmark suite scores with error bars showing the standard error. The results indicate that for all benchmarks except the Earley-Boyer benchmark, V8 running on the EbbRT Library OS slightly outperforms V8 running on Linux. The results show a difference in mean total score of 259.47 ± 7.33, demonstrating that the benchmark running under EbbRT shows a $1.74\% \pm 0.049\%$ improvement in mean total score.

These results illustrate that a library OS can support a

managed runtime in a fashion that achieves equivalent results to a general purpose OS. In fact our straightforward port achieves a performance advantage as compared to Linux. However, a library OS can provide further opportunities to achieve better performance through customization of low-level system software. For example, one might explore modifying V8 to take advantage of direct access to page tables for garbage collection.

While running these benchmarks within VMs, we observed that the total execution time of the EbbRT VM was about 10% less than the Linux VM. EbbRT's smaller footprint and faster boot process allowed the VM to run the benchmark and terminate more quickly. This effect is explored in detail in the Sage use case.

5.2.2 Discussion

This use case demonstrates that it is possible to get a complex runtime to execute on EbbRT along with the large body of software it supports. In the memcached scenario, we demonstrated that EbbRT benefits applications by allowing them to map more closely to the hardware. However, our memcached implementation was written directly to our base interfaces. Many applications are considered to be too large to be completely rewritten to target EbbRT, despite potential performance improvements. The node.js result demonstrates that EbbRT's performance advantages can be realized by a large body of existing software.

The port of node.js (including V8 and libuv) is 1585 lines of code of which the majority (1237) is in the port of libuv. The port took a single graduate student two weeks to bring to level of completion where we were able to run node.js webservers capable of serving files (exercising both networking and file access interfaces). The final boot image which is generated is 5.76 megabytes in total size.

A key result of this port is the ability to run complex applications without requiring modification to the system's base layers. The node.js application uses the same *Event Manager* and *Networking Manager* as the memcached application. We found no need for pre-emption while porting this application. This provides evidence that our approach leads to constructing reusable software, without which the effort to port applications to EbbRT would be daunting.

This use case, along with memcached, shows that the primitives provided by EbbRT are simple and lightweight, allowing for low level optimizations, yet the same primitives are also expressive enough to be suitable for a wide range of different applications.

5.3 Sage

In this case study we extend Sage (mathematics software)[1] with EbbRT. This study demonstrates how a process running on a general purpose OS can elastically exploit an IaaS by offloading functionality to specialized library OSs.

Sage is a large open source mathematics environment similar to Matlab. It provides many common math library routines and objects through a Python interface (typically accessed via an interactive shell). Sage does supports MPI interfaces but this puts the burden on a mathematical user to write explicit parallel code and requires users to setup a dedicated static MPI cluster. EbbRT integration into Sage provides a path to using IaaS resources to transparently enable a user to do large scale parallel computation with no additional burden.

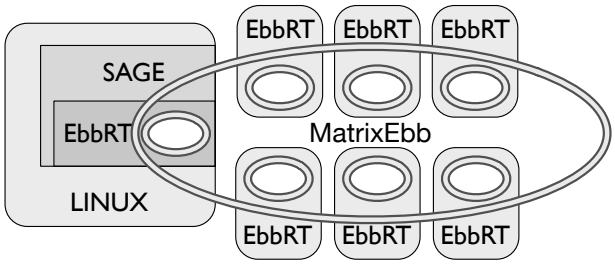

Figure 7: EbbRT Sage Matrix Integration.

Following the standard method for adding an extension to Sage, we created a Python module which can be dynamically loaded into the Sage environment. This module links with the EbbRT Linux library and provides a python matrix object which wraps a matrix Ebb. When this python matrix is instantiated at the command line, an instance of the matrix Ebb is constructed to back it. After calls are made to the python matrix object, they are forwarded to the matrix Ebb which may internally distribute its functionality to satisfy its interface. Figure 7 illustrates the realized runtime structure. Using our experimental IaaS infrastructure, the Ebb matrix behaves as we expect spawning and destroying VM's and distributing the work across them.

In our particular matrix Ebb implementation, the representative running within the Sage process on Linux allocates nodes from the *Node Allocator* booted with the EbbRT Library OS to hold a fixed tile of the matrix values and perform the core computations on that matrix tile. The matrix Ebb links with the Boost uBLAS library to provide local matrix operations. Nodes are allocated lazily, when an operation requires a particular portion of

the matrix for the first time. This structure allows for matrix operations to be done both lazily and in parallel. Our matrix Ebb implements a number of matrix operations such as summation, multiplication, element-wise randomization, and element access.

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

5.3.1 Evaluation

Parallel matrix computation is extensively studied in High Performance Computing (HPC). The operations being done, the algorithms for the operations, the data distribution and interconnect technology are all important factors in deciding how to structure and implement a distributed matrix. The performance of the matrix and its operations that we have implemented as an Ebb is what one would expect from a straight-forward distributed matrix implementation. Portions of an operation that are tile specific are done on the appropriate node and cross tile portions of an operation induce communication. As IaaS data-center inter-connects continue to evolve we expect that there will be a broad spectrum of choices on how one might implement such an Ebb matrix.

Independent of any particular matrix Ebb implementation, the value of EbbRT is its ability to enable a developer to stitch-in the node allocation and deallocation into Sage, and the ability to maximize the value of an allocated node. The latter case breaks this down along three dimensions: 1) the latency in utilizing a newly allocated node (booting and dispatching application logic), 2) the efficiency in executing the compute intensive work, and 3) ability to maximize the I/O capabilities of the node for data exchange. The last of these three was explored in the memcached use case which illustrated the ability to customize I/O paths. We focus our Sage evaluation on the first two dimensions.

In these experiments we construct and study two bootable images, one EbbRT based and the other Linux. In the case of EbbRT we package the back-end EbbRT Sage application matrix code along with test logic. Using the Ebb Matrix code the test logic allocates a local matrix tile of a specific size and then performs a set of local only actions on the tile values. Specifically, the operations include zeroing the values (as per the requirements of Sage), randomizing the values and obtaining a sum across all the values of the tile. This behavior is repeated for a specified number of iterations and then the VM is powered down via appropriate firmware calls. The EbbRT test code is carefully constructed not to do any external communication. However, it does go through its normal boot process as a backend which includes initializing the network interface card and protocol stack.

In the case of Linux we build an EbbRT Linux application that runs the same back-end EbbRT application matrix code, however, it uses the hosted version of the EbbRT library to run on Linux. This application includes the same test logic as above and provides us with a comparable application workload.

While IaaS providers have different image deployment strategies, one factor that will affect boot latency is the time to transfer the boot image to the node. In our case, the EbbRT image is an uncompressed bootable ELF image that is 1.2 Megabytes in size, while the Linux boot image is in aggregate 31.6 Megabytes in size. It is composed of a compressed Linux kernel of 5.6 Megabytes, the generic initial compressed ram disk of 24 Megabytes and the test application (including the libraries it depends on) that increases the compressed ram disk by an additional 2 Megabytes. These sizes imply a significant difference in potential transfer times – 240ms vs 10ms at 125Mb/s (ideal 1gE) and 24ms vs 1ms at 1250Mb/s (ideal 10gE). While we did not aggressively optimize the size of the Linux inital ram disk or kernel, we believe our values to be reasonable while maintaining the systems generality and robustness.

	Linux	EbbRT	Ratio
Instructions	1915M (0.1M)	73M (0.07M)	0.038
Cycles	1711M (0.7M)	95M (0.06M)	0.056
Seconds	4.01 (0.01)	1.52 (0.02)	0.376

Table 1: Sage Boot Costs.

To obtain a baseline measurement, we ran the Linux and EbbRT images with the application configured to do no work. This gives the base costs for the two configurations to boot and initialize the VM, launch the application and terminate. Using the Linux perf tool we obtain aggregate elapsed time in seconds for the VM along with guest cycles and instructions. We report mean values along with standard error in the mean. From the data shown in Table 1, we see that EbbRT conducts dramatically less work within the VM and boots approximately 2.6 times faster. It is critical to note that the guest cycles and instructions

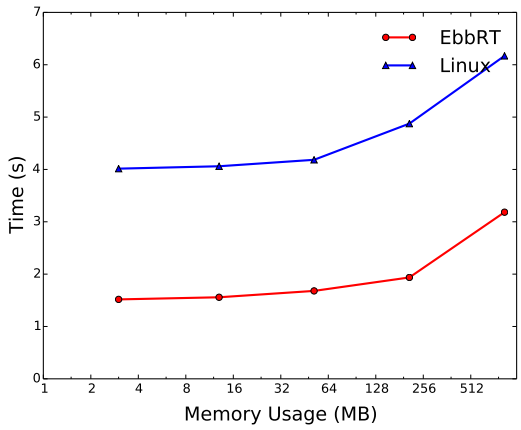

Figure 8: Aggregate time of the VMs lifetime

only reflect work done in the VM and not work done in the hypervisor on behalf of the VM. However, the elapsed time reported by perf is the entire time the VM was alive. Given this, we suspect that the true VM efficiency is not solely reflected in the IPC computed based on the VM guest instructions and cycles.

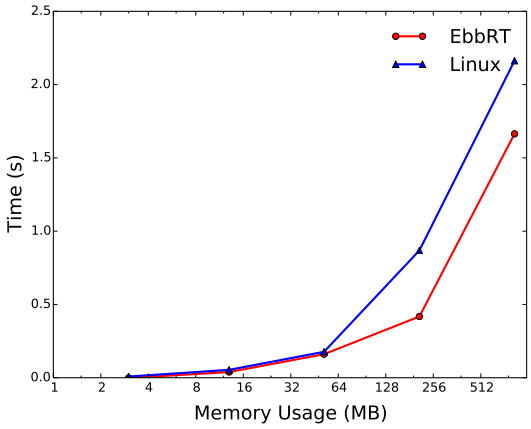

Figure 9: Work time of matrix operations

Figures 8 and 9 present the data we obtained from a series of experiments that vary the matrix size. As the matrix size increases, the elapsed time grows proportional to the matrix size, as shown in Figure 8. EbbRT demonstrates a smaller elapsed time for all sizes. At the largest size Linux takes 6.17 seconds vs 3.18 seconds for EbbRT (1.94 times faster). While the reduced boot costs of EbbRT are a dominant factor in the result, it is not sole factor.

Subtracting the boot times (Table 1) we can isolate the application work phase, as shown in Figure 9[1]. From this curve we see that EbbRT does the same application work in a smaller elapsed time. We suspect that this is due to EbbRT placing fewer demands on the hypervisor through its use of large pages and simplified address space structure. Doing so will require fewer guest page-table accesses and attendant hypervisor processing for their maintenance. While not shown here, the application work in terms of instructions and cycles is roughly the same for both Linux and EbbRT.

5.3.2 Discussion

The EbbRT Linux front-end library made the integration of IaaS resources into Sage equivalent to developing any other extension to Sage. The Sage integration code we developed (143 lines) remained stable and independent to any changes we made to the Ebb Matrix functionality itself or optimizations of the backend operations.

From the data, we see that EbbRT benefits from its library OS architecture to achieve greater efficiency both in initializing the node and in doing the actual computational work. This latter result is surprising, as the application is fundamentally a tight user loop and computationally bound. However, it is still memory intensive and address space maintenance can induce overheads that will increase the elapsed time. In general, a library OS running a dedicated application can eliminate overheads associated with general purpose OSs that induce noise that slow down computational and memory bound applications. This result is in line with observations made in the HPC community.

While the results in this evaluation are focused on local per-node performance, an important opportunity that EbbRT enables is the ability to customize the Matrix implementation. Specifically, one could exploit direct access to the interconnection network and the unique hardware facilities that it offers. As was demonstrated in the memcached use case, the matrix processing could be integrated into the interrupt handling. This is particularly attractive as one could eliminate standard protocol processing and exploit both hardware and software techniques that are typically used in HPC, such as RDMA and hardware support for combine and collective operators.

6 Conclusion

We have introduced a new system software runtime called EbbRT. EbbRT explores a unique system architecture,

[1] In this data we observed a single outlier which has been removed.

where general purpose OSs are augmented by small library OSs to exploit the features of an IaaS provider. Our system adopts a non-preemptive execution model which allows the event driven nature of modern cloud applications to take advantage of the hardware directly. We also explore a new partitioned object model, called Ebbs, which encapsulate distributed software, allowing components to be independently customized and reused.

Our runtime allows applications to run software on our lightweight library operating system without requiring large investment in porting existing, non-performance critical functionality. We have demonstrated through our memcached implementation that by allowing applications to more directly exploit the hardware, significant performance advantages can be realized. Our node.js port shows that by offloading functionality, we can rapidly port rich applications to reap the benefits of library operating systems. Finally, our Sage application shows how we can integrate our library with existing applications to enable the use of IaaS resources in a fine-grain fashion.

In contrast to a conventional operating system, which at some level can be defined to be complete, EbbRT is intended to provide a structure for constantly evolving system software to meet new application needs and hardware. Results presented in this paper give us some confidence that the architecture will be flexible enough to meet this challenge.

Serious open questions remain about our system design. One important assumption of this work is that IaaS providers will further improve the ability to rapidly provision hardware on demand. We fear that some value of our system will be lost if this does not bear true. Another significant concern is that the development of different applications will lead to large vertical stacks of software which do not compose. Many different implementations of system software may also cause a significant configuration challenge. It remains to be seen how these challenges impact the system.

References

[1] Sage.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[3] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. 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.

[4] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience distributing objects in an smmp os. *ACM Trans. Comput. Syst.*, 25(3), Aug. 2007.

[5] H. Bal, M. F. Kaashoek, and A. S. Tanenbaum. A distributed implementation of the shared data-object model. In *IN USENIX WORKSHOP ON EXPERIENCES WITH BUILDING DISTRIBUTED AND MULTIPROCESSOR SYSTEMS*, pages 1–19, 1989.

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

[7] G. Brun-Cottan and M. Makpangou. Adaptable Replicated Objects in Distributed Environments. Research Report RR-2593, 1995. Projet SOR.

[8] A. Dunkels. lwip - a lightweight tcp/ip stack.

[9] 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.

[10] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, Aug. 2004.

[11] B. Gamsa, O. Krieger, J. Appavoo, and M. 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, Berkeley, 1999. USENIX Association.

[12] Google. V8 javascript engine.

[13] P. Homburg, M. V. Steen, and A. S. Tanenbaum. Distributed shared objects as a communication paradigm. In *In Proc. of the Second Annual ASCI Conference*, pages 132–137. University Press, 1996.

[14] Joyent. Node.js.

[15] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. 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.

[16] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. 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.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[18] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. 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.

[19] Y. Li, R. West, and E. Missimer. A virtualized separation kernel for mixed criticality systems. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 201–212, New York, NY, USA, 2014. ACM.

[20] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. *SIGPLAN Not.*, 48(4):461–472, Mar. 2013.

[21] M. Makpangou, Y. Gourhant, J.-P. Le Narzul, and M. Shapiro. Fragmented Objects for Distributed Abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, 1994.

[22] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 153–167, New York, NY, USA, 1996. ACM.

[23] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. 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, Oct. 2014. USENIX Association.

[24] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. 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 '11, pages 291–304, New York, NY, USA, 2011. ACM.

[25] B. Rhoden, K. Klues, D. Zhu, and E. Brewer. Improving per-node efficiency in the datacenter with new os abstractions. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 25:1–25:8, New York, NY, USA, 2011. ACM.

[26] M. Shapiro. SOS: A Distributed Object-oriented Operating System. In *Proceedings of the 2Nd Workshop on Making Distributed Systems Work*, EW 2, pages 1–3, New York, NY, USA, 1986. ACM.

[27] R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *J. Supercomput.*, 9(1-2):105–134, Mar. 1995.

[28] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 230–243, New York, NY, USA, 2001. ACM.

[29] D. Wentzlaff, C. Gruenwald, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An Operating System for Multicore and

Clouds: Mechanisms and Implementation. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 3–14, New York, NY, USA, 2010. ACM.

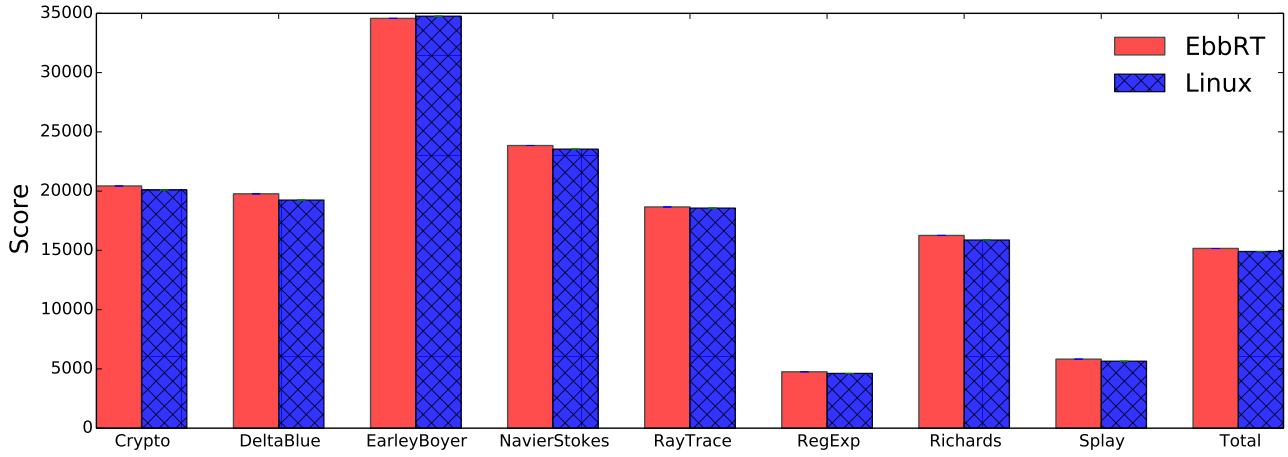

Figure 6: V8 Benchmarks