EbbRT: Elastic Building Block Runtime - Overview

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

1 Introduction

Infrastructure as a Service (IaaS) provides a developer the ability to construct applications that dynamically acquire and release potentially large numbers of raw virtual or physical machines (nodes). The Elastic Building Block Runtime (EbbRT) is a new runtime for constructing and retro-fitting applications to take advantage of this opportunity.

EbbRT is a realization of the *MultiLibOS* model[**?**]. This model is based on the simple idea that not all IaaS nodes, used by a single application, need a general purpose OS. Rather, an asymmetric system software structure can be adopted where an application is distributed across a mix of general purpose OSs and specialized library OSs. The general purpose OS nodes support complete OS functionality and legacy compatibility. The rest of the nodes execute simple, customized, library operating systems that support a single application operation.

EbbRT provides a lightweight runtime that enables the construction of reusable, low-level system software which can integrate with existing, general purpose systems. It achieves this by providing a library that can be linked into a process on an existing OS, and as a small library OS that can be booted directly on an IaaS node.

The two core primitives that EbbRT provides are:

Events: A lightweight, non-preemptive execution model that allows for an event-driven programming style to map directly to hardware interrupts.

Elastic Building Blocks: An object oriented programming model that separates interface design from distributed implementation which enables the construction of composable and reusable software.

Context for EbbRT and our choice of primitives is presented in section 2. Section 3 provides an overview of EbbRT's architecture and describe our prototype. Section 4 evaluates the prototype using three use cases, namely: 1) an EbbRT implementation of memcached[15], 2) a port of the V8[17] javascript engine and node.js[20], and 3) the integration of an elastically allocated distributed matrix object into the Sage[1] environment. These use cases demonstrate that EbbRT:

A. enables applications to achieve high performance by customizing low-level system software,

B. can support rich complex applications and run times, and

C. allows an application to be modified incrementally to exploit the elasticity and scale of an IaaS.

2 EbbRT Context

EbbRT's value and novelty lies in its unique combination of ideas from prior work. Specifically, EbbRT draws from work on library OSs, event driven software, and the use of partitioned object models in both multiprocessor and distributed systems software construction. In this section we provide the context for each and state how it is realized in EbbRT.

2.1 Library OSes

Library operating systems[14] organize a single application and the OS functionality it requires into a single address space and protection domain. The application code directly links 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 [9, 8, 10, 26, 32, 22, 11, 30, 25, 29, 3]. 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[32, 10] or API[26, 22, 25, 30, 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, 22] and Ocaml[26] and/or 3) using a shim layer to forward system calls to a 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's 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 and front-end process.

EbbRT exploits library OSs for the back-ends to allow 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 a set of manually constructed functions that invoke 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. While labor intensive, this approach to compatibility is tractable for supporting managed runtime environments as demonstrated by our port to the node.js runtime.

Other library OSs [3, 26] have been developed to be deployed on a cloud, but EbbRT is the first distributed library OS we are aware of. Other research groups exploring new operating systems for the cloud [34, 39] 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.

2.2 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 lowest level software 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[38, 23, 28].

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 anonymous functions 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 [37, 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 schedulers to be constructed [4].

2.3 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-resource (object) basis. Fragmented Objects(FOs) [12, 27, 35] and Distributed Shared Objects(DSOs) [7, 18] 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[16, 24, 5] demonstrated the effectiveness of a partitioned object model in the construction of multi-processor 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.

3 Architecture and Prototype

In this section we present the architecture of EbbRT and our prototype of it. The final subsection discusses an example code fragment from our prototype to clarify and illustrate salient concepts and features.

3.1 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

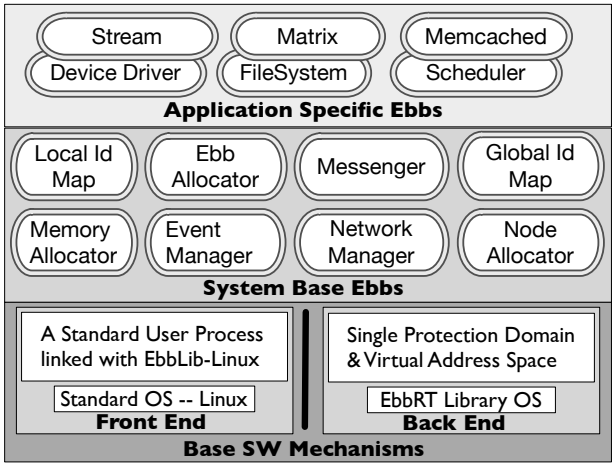

Figure 1: EbbRT Architecture

lowest layer provides the base software mechanisms for constructing the address spaces that the Ebb application will run within. The rest of the EbbRT software, illustrated in the top two layers, takes the form of Ebbs and execution occurs on light-weight 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 par-

ticular 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 and back ends.

3.1.1 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. Back-ends, however, 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. This includes, for example, arbitrarily large virtual memory regions to be used for stacks, core-specific memory regions, and the ability for applications to specify their own fault handler for these regions. A full discussion of the base software mechanisms is out of scope for this paper

3.1.2 System Base Ebbs

The EbbRT system is fundamentally composed of a set of Ebb instances that provide interfaces that additional Ebbs can be developed to. A single well known static instance of each of these Ebbs is provided when the application is initialized. In general, 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. Below we briefly describe the role of each.

Memory Allocator: On back-end nodes after the base EbbRT Library OS initializes the virtual memory subsystem the *Memory Allocator* is initialized to serve as the general purpose memory allocator. The C and C++ runtimes are configured to use it.

Event Manager: This Ebb is responsible for providing the event interfaces and implementing a basic non-preemptive event loop per-core. It does not provide threads but rather a set of interfaces for specifying callbacks to execute in response to hardware events, in the form of interrupts, and software events. Software events are function calls that can, but do not have to, execute asynchronously with the caller. Events run to completion unless they manually switch stacks via calls to Event Manager methods.

Ebb Allocator and Local Id Map: These two Ebbs provide the base support for additional Ebbs to be created. On each node the the *Ebb Allocator* manages a range of EbbIds that are known to be unique and can be used to identify a single instance of an Ebb. The *Local Id Map* can be used by an Ebb instance to store common data to all of its representatives on a node.

Network Manager and Messenger: These two Ebbs provide a basic set of communication facilities so that cross node communication is possible. In particular, the *Network Manager* provides an event driven interface to the networking facilities of the local node. The *Messenger* uses the *Network Manager* to provide an interface for sending a message to an Ebb on a particular node. The arrival of a message can cause a representative to be created on that node.

Global Id Map: This Ebb provides an application wide table that serves as a place for Ebb instances to store data accessible across all nodes of the system. When an Ebb is instantiated, it can place data into the *Global Id Map* that can be used to construct representatives on an arbitrary node. In particular when an Ebb is first accessed on a node and it can not find information in the *Local Id Map*, the Ebb can then consult the *Global Id Map*. The data obtained can be used to populate the accessing node's *Local Id Map* which in turn can be used to construct representatives of the Ebb on that node as needed.

Node Allocator: This Ebb provides an interface to the rest of the Ebb software for acquiring and booting a node. Its implementation is specific to a particular IaaS's interfaces and performance characteristics. While on a commodity IaaS it can take tens of minutes to provide a node to a client, at least one IaaS is capable of providing hundreds of physical nodes to a client in sub-second time frames[6]. The *Node Allocator* can bridge this gap by internally creating a free pool of pre-allocated nodes loaded with a special image that puts the node into a dormant state waiting to be released for application use.

3.1.3 Application Specific Ebbs

Above the base EbbRT 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 and its fine grain decomposition provides many degrees of freedom to customize even its most basic functionality such as the event processing loop and interrupt dispatch by implementing an application customized implementation of the

Event Manager. Keeping with this compositional theme, EbbRT expects many traditional features of a library OS to be provided as independent libraries of Ebbs. This includes things like additional device support, files, network protocols and abstractions. Similarly, the enablement of libraries of application Ebbs that provide application specific Ebbs such as scalable and elastic matrices, are a core value of the architecture. 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.

3.2 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 custom EbbRT backend 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 tool chain includes a version of the C and C++ standard libraries.

In order to explore EbbRT, we have constructed a simple synthetic IaaS that launches KVM instances. 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. In our prototype, the *Node Allocator* is a simple implementation that just calls out to our IaaS daemon. This daemon launches KVM virtual machines to boot with the specified image and set of arguments.

The *Global Id Map* of our prototype is a very simple centralized implementation where the representative on the launching front-end maintains the entire hash table. We expect other implementations of the *Global Id Map* to take on a much more robust and complex structure. In some scenarios a Chord [36] or Zookeeper [19] based implementation is likely to be appropriate.

The *Memory Allocator* implementation has been implemented based on the SLQB design[31]. It is naturally realized as an Ebb given its per-core design. We choose SLQB for its multi-core and NUMA friendliness and expect it to be a good match for multi-core optimized Ebbs. It is perfectly reasonable, however, for alternative *Memory Allocator* implementations to be developed as the need arises.

As illustrated by our case studies (see 4.1) our prototype *Event Manager* is simple but effective. We expect that a wide range of *Event Manager* variants be useful in tightly tailoring the event loop to an application's specific needs beyond the ones that we have explored so far. While our prototype implementation lacks preemption and threading, our design allows for the *Event Manager* implementation to be evolved to serve as the foundation layer for scheduler activation[4] inspired Ebb libraries that provide pre-emption and threading as needed.

The *Network Manager* and *Messenger* implementation in our prototype have been deeply influenced by our current use of traditional Ethernet and IP based communication.

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

3.3 Example

We conclude our discussion of the EbbRT architecture and our prototype with an example. The top of Figure 2 presents a fragment of code that spawns a new software

```
ebbrt::event_mgr->Spawn(
      // anonymous function as argument
      [](){
            ebbrt::kprintf("%lld %lld %lld\n",
                           event_mgr->GetNum(),
                           event_mgr->GetNumNode(),
                           event_mgr->GetNumCore());
          }
    ); // close bracket of Spawn call
```

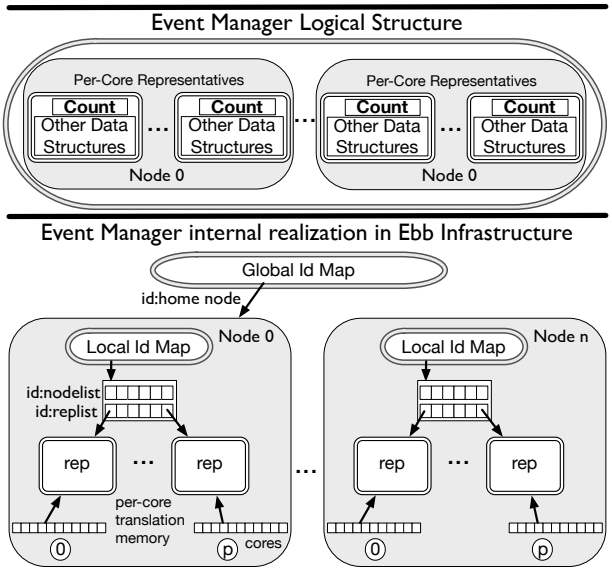

Figure 2: EbbRT Example

event to be executed. To do so, the `Spawn` method of the *Event Manager* Ebb is invoked. This code fragment can be run in any EbbRT address space, front or back end, of an application.

The *Event Manager* is accessible via the global symbol `ebbrt::event_mgr`. Its value is the EbbID of the single statically provided *Event Manager* instance. Every Ebb instance must have a unique EbbID that is obtained either statically or dynamically from the *Ebb Allocator*. The *Ebb Allocator* manages the EbbId values to ensure uniqueness. In our prototype, we utilize the C++ support to override the dereference operator ($\rightarrow$) to invoke logic that translates the EbbID to a representative pointer.

The middle of Figure 2 illustrates the logical structure of the *Event Manager*. It is a fully replicated Ebb having one representative per-core, per-node. When the `Spawn` method is invoked the Ebb's dereference operator automatically invokes it on the representative associated with the core executing the call. The bottom of Figure 2 illustrates how this structure is realized by the *Event Manager* implementation using the Ebb infrastructure.

Within each EbbRT address space is a region of per-core translation memory that appears at the same virtual address but is backed by per-core physical memory[1]. An EbbId translates to an offset in the translation memory region. On each core, the location in the translation memory associated with the *Event Manager*'s EbbID caches a pointer to the core specific representative. The *Event Manager* uses the *Local Id Map* to map it's EbbID to a single instance of a node specific data structure. In the case of the *Event Manager* the data structure contains two maps, a `replist` and a `nodelist`. The former is a master list of representatives that exist on a node and the cores they map to. And the latter records the network addresses of all the nodes on which the *Event Manager* has representatives. Finally, the *Event Manager*, during initialization of the application, places in the *Global Id Map* the network address of a node that serves as the *home* node. The *home* is responsible for maintaining the master `nodelist` which is cached as needed on the other nodes.

Using this logical structure and layout in the Ebb infrastructure, the representatives of the prototype *Event Manager* implement the *Event Manager* interfaces. In the code fragment, we see the `Spawn` interface being invoked. This method specifies a function to be dispatched as an event. The default *Event Manager* behavior is for this function to be synchronously invoked on its own stack while the caller's stack is placed aside. If this function or any of its subsequent functions attempt to block via other methods provided by the *Event Manager*, the calling stack will be reinstated and execution of the caller will be resumed at the return of `Spawn`. In this fashion, the *Event Manager* implements a form of manual handoff scheduling to spawned events. From the programmer's perspective, however, spawned functions should be assumed to be asynchronous with respect to the caller of `Spawn`. Explicit spawn interfaces are provided that allow the programmer to force the function to be asynchronous in which case the *Event Manager* will place it on a list to be dispatched from the *Event Managers* event loop when execution returns to it.

In the case of the example code listed, the function to be executed is an anonymous function or more precisely a C++11 lambda. The syntax allows the code for the function to be specified in-line and provides the ability to create a closure that captures values that are in scope when the `Spawn` method is invoked. The EbbRT prototype makes frequent use of lambdas to simplify continuation based programming. In this case the function to be invoked calls three methods of the *Event Manager*; `GetNum`, `GetNumNode` and `GetNumCore`, these respectively return the number of functions that have been dispatched across the entire application, on this node and on this core by the *Event Manager*.

By exploiting a fully replicated structure, the *Event Manager* representatives by default implement the `Spawn` and event loop on a per-core basis using its own member data structures. As part of dispatching functions, the representative maintains a counter that it increments and thus tracks the number of functions called on this core. Given the non-preemptive nature of execution and per-core representative structure operations on these counters do not need to synchronized. Additionally features of the memory allocator are used to ensure that the data structures of each representative are on distinct cache lines. This ensures that there is no false sharing between the counters, and thus good performance and scalability will be achieved on event dispatch while accurate counts are maintained. To implement the various `GetNum` operations the representatives do gathers as necessary using the `replist` and `nodelist` maintained on each node, and sending messages as necessary.

There are of course several alternative approaches to organizing representatives in the infrastructure. For example rather than, or in addition to, maintaining master `replists` and a master `nodelist`, the representatives could contain pointers that link them in to a ring and similarly the nodes could contain network addresses of neighbors to also form a ring. This flexibility exists to allow Ebbs to utilize the infrastructure in a manner that is most

[1]On front-ends thread local storage (TLS) is exploited to provide similar function albeit with greater overhead.

appropriate for its needs, the application it designed to serve, and features of IaaS interconnection networks.

4 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 rich managed runtime and demonstrates the viability of supporting rich 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.

4.1 Memcached

This case study describes a memcached[15] server, implemented with EbbRT, to produce a bootable image. This use case demonstrates that EbbRT can be used to enable very simple application code to fully exploit the (virtualized) hardware and illustrates the use of the event driven execution model for a 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 optimisation of networked systems. It has also been shown by previous work to incur significant OS overhead [21], and hence is a natural target for a library OS.

4.1.1 Implementation

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 [40]. To a developer with knowledge of the EbbRT interfaces, this basic application can be developed in a single afternoon.

The upper portion of Figure 3, 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 uses one call down into 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-

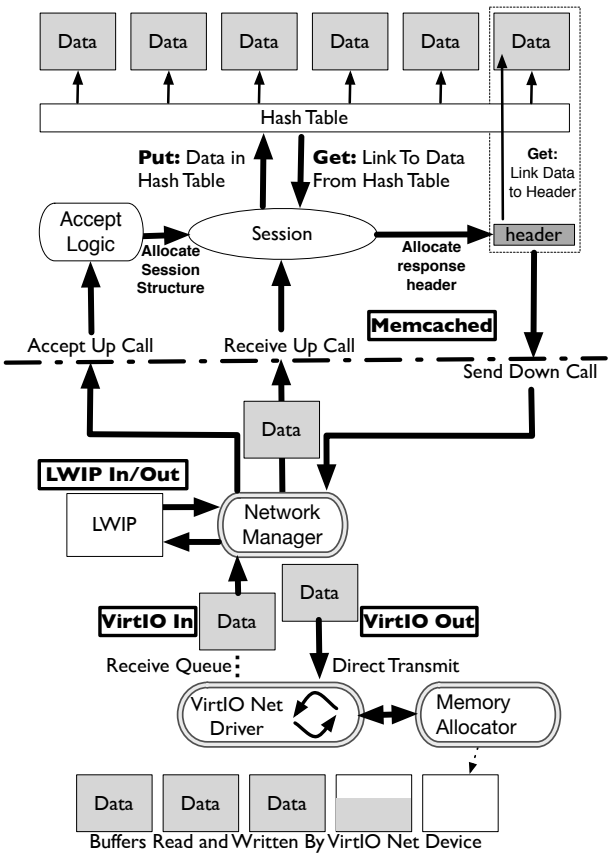

Figure 3: EbbRT Memcached Application

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 3 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) use a re-occurring *idle* event, this event runs when no other events exist. This poll event inspects the device state to check for used buffers. If none are found, interrupts are enabled on the device and the idle event handler is unregistered. 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) [13] 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 completion (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. The event loop will disable interrupts and then execute the idle handler.

Jointly the upper and lower portions of Figure 3 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. So much so that the application hash table directly stores buffers that were originally allocated by the device driver when a *put* is invoked. And when *get* is invoked this same memory can be chained along with a newly allocated message header for direct transmission by the device.

4.1.2 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 [2].

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 4 shows the throughput of memcached for a small payload as we increase the number of concurrent requests from the client. We see that 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.

With more than 64 connections, we then see that Linux's throughput maintains the same rate, while EbbRT's gradually degrades. The source of degradation for EbbRT is shown in Figure 5. We see that the main degradation is in the performance of the LWIP receive performance. Examining the code, we find 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 we hit a concurrency of 256, because the number of concurrent packets exceeds the ring buffer of the device, resulting in TCP retries, and slowing down some clients resulting in the LWIP optimization having less of a negative consequence.

This figure also demonstrates another important point. Even in our 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 6 shows the performance of memcached, with a fixed concurrency of 64 sockets (our peak) as we increase the payload size. We see that 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.

Discussion We see from the performance data above that we are 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

[2]The vhost-net kernel module provides improved network performance through in-kernel network packet hand-off.

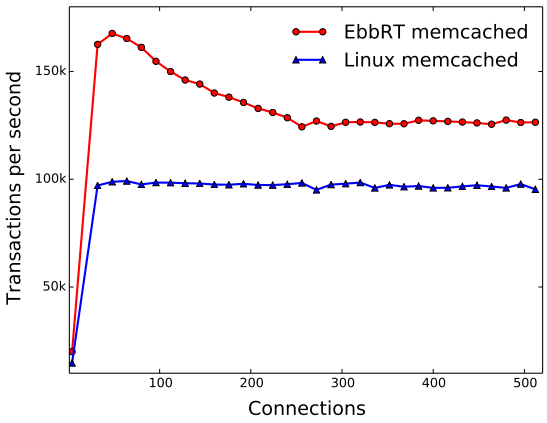

Figure 4: Memcached throughput as a function of number of concurrent clients

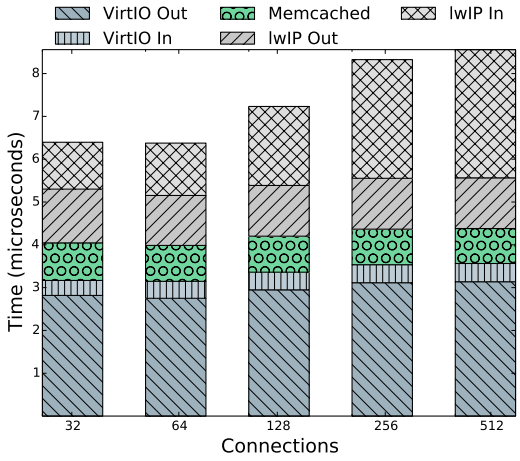

Figure 5: Per-request latency breakdown of EbbRT memcached

smaller packet size. The current performance degradation is due to the LWIP library we use 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. As with other systems [26], we expect to need to implement/port a more performant TCP implementation over time.

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

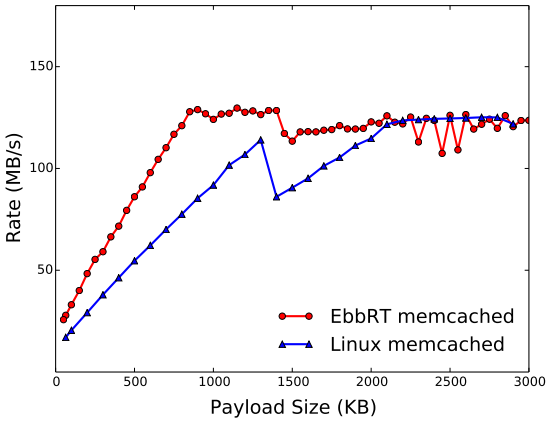

Figure 6: Memcached throughput as a function of payload size

every client request.

One option with an EbbRT implementation is to make optimizations that are brittle in that they 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 very brittle change, that only performs well for just one application, is a reasonable option in a system like EbbRT.

The results we have obtained are consistent with Chronos[21], 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 we can easily provision nodes on an IaaS cloud, we believe the EbbRT design provides a natural alternative to supporting the performance demanding applications that are a poor match for our general purpose systems, and alleviate the burden on commodity OSs to be simultaneously general purpose and robust while also needing to be special purpose and customized for a single application.

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

Our experience developing memcached also demon-

strates that EbbRT is a natural match for the intrinsic event driven nature of this kind of application. Applications like memcached, which are fundamentally about handling networking events, are a mismatch for general purpose OSes, on which they have to construct an event model 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 efficiently support multi-core applications, the LWIP library we are using limit our multicore performance. Removing this barrier is, for now, future work.

4.2 NodeJS

This case study describes the port of node.js, a javascript environment for server-side applications, to EbbRT. It illustrates three points: 1) That EbbRT can support complex managed code environments, allowing existing software to run unmodified on the library OS back ends. 2) How EbbRT's non-preemptive, event-driven execution environment is suitable for even large, complex applications such as node.js. 3) That 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 we reached in the process of running various Node.js applications.

Libuv manages an event loop which dispatches callbacks installed by the application. The application must execute the `uv_run` function to begin dispatching events. In the Linux implementation, installing a callback for reading data from a tcp socket is implemented by informing Linux of the desire to be notified when the tcp socket is available for reading, this is done via the `epoll` system call. In contrast, the EbbRT implementation installs a callback to be invoked by the Network Manager when tcp data arrives. The implementation of `uv_run` must then save its context (stack and a few general purpose registers) to be woken up when the callback is invoked. However, this callback executes on a separate stack from the `uv_run` stack. The callback can synchronously activate the previously saved context to execute the callback handler installed by the application. These context switches are much simpler than equivalent context switches on general purpose OSs because they do not involve a protection domain crossing and due to the lack of pre-emption, do not have to save many registers (only those that are callee saved as mandated by the ABI). This allows the libuv callbacks to be invoked synchronously from the hardware interrupt that caused it in much the same way that the memcached application was able to.

We were able to implement the networking interfaces provided by libuv in this fashion by installing callbacks to reactivate the `uv_run` context and invoke the application callbacks. This was sufficient to allow us to run node.js applications including tcp stream processors and web servers.

Filesystem access was implemented by invoking a *FileSystem* Ebb linked into the application. Rather than implement a file system and hard disk driver, our implementation offloaded calls to a representative running in a Linux process. Specifically our implementation of Libuv invokes the *FileSystem* Ebb which performs the offload by sending messages between its representatives. Our implementation of the *FileSystem* Ebb is naïve, sending messages and incurring round trip costs for every access rather than caching data. This allowed us to quickly get the rich functionality provided by the Linux filesystem with minimal development effort. Implementing a caching layer would require only changes to the *FileSystem* Ebb, without modifications to libuv.

Offloading allows us to execute node.js by launching a Linux process linked into the EbbRT library which then allocates a node loaded with the EbbRT library OS linked to node.js. Node.js can, via an Ebb, read the command line arguments that were originally passed to the Linux process. This then indicates the filename of the node.js script which is fetched from the Linux process and then loaded. This model allows us to rapidly spawn node.js instances on their own machine with integration via the frontend file system.

4.2.1 Discussion

In the memcached scenario, we demonstrated that EbbRT benefits applications by allowing them to map more closely to the hardware. Our memcached implementation was written directly to our base interfaces. Many applications, however, are too large to consider completely rewriting to target EbbRT, despite potential performance

improvements. Node.js offers us a number of different layers to consider for porting. One could have ported at the system call layer and emulated Linux and linked directly to the Linux libuv implementation. The Linux libuv implementation in many cases uses a thread pool to make blocking system calls in cases where Linux has poor support for non-blocking interfaces. EbbRT can support the libuv interfaces more directly and so we opted to do the port at that layer. This illustrates EbbRT's suitability for an increasingly popular programming paradigm. Many cloud applications such as nginx [33], memcached, node.js, are designed to be event driven and depend on various event libraries to abstract the OS interfaces designed for event dispatch. EbbRT natively provides these interfaces and provides natural mappings for these applications.

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.

Had we needed to construct an execution environment for node.js which was orthogonal to the environment used by memcached, it would be difficult to argue that our approach is practical. The software written for one environment would not be able to interact with the software in another. The primitives provided by EbbRT are simple and lightweight, allowing for the optimizations exploited in the memcached application, yet the same primitives are also expressive enough to be suitable for a wide range of different applications.

From a networking perspective, node.js running on EbbRT has the same kind of performance advantages as in the memcached use case described above. This work also opens up the door for other performance advantages in optimizing the managed runtime to take full advantage of direct control of the page table and event dispatching for sake of improved garbage collection, memory management, and thread management [3, **?**].

4.3 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 an open source mathematics environment similar to Matlab. It provides many common math library routines and objects through a Python interface (typically accessed via an interactive shell). One limitation of Sage is that all of its standard routines and objects are designed to execute on a single machine and do not scale. Sage does support 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 for using IaaS resources to transparently enable a user to do large scale parallel computation with no additional burden.

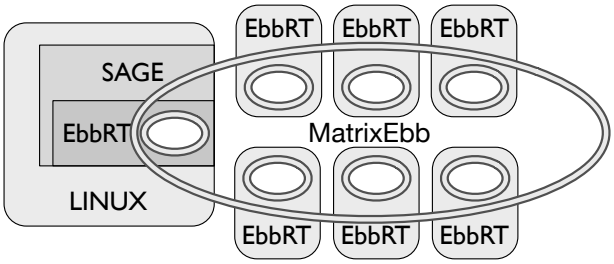

Figure 7: EbbRT Sage Matrix Integration.

Sage incorporates many software libraries; porting the entire Sage environment would be a significant investment in developer time. We instead, explore the ability to use EbbRT to incrementally modify existing applications.

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. When 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.

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. For example, as matrix elements are set, the Linux representative will allocate nodes as necessary to store the tile of the matrix that the element belongs to. Operations of the matrix Ebb such as element-wise randomization can naturally be done in parallel across the tiles. Our matrix Ebb implements a number of matrix operations such as summation, multiplication, element-wise randomization, and element access.

The EbbRT Library OS is well suited for offloading computationally expensive functionality because it allows the application complete control of the hardware. For example, interrupts are disabled which prevents context switches from causing cache pollution to slow down the computation. Additionally, complete control over memory allows the use of large pages to reduce TLB contention.

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

4.3.1 Discussion

The performance of the matrix operations that we have implemented as Ebbs is what one would expect from a distributed tile oriented matrix implementation. Fundamentally, the point of this exercise was not a demonstration our particular matrix Ebb's parallel superiority but rather in EbbRT's ability to extend Sage with a distributed matrix Ebb and have it naturally and transparently used.

EbbRT's MultiLibOS design and implementation enable a customized version of a front-end library and back-end library OS to be developed that targets a core application function that can be integrated in to an existing complex application stack. Our implementation was able to introduce fine grain elasticity into Sage where IaaS node consumption grows and shrinks with not only the number of matrix instances but even more finely with the active tiles of large matrices.

This study illustrates that EbbRT can be an effective tool for evolving the use of IaaS resources. Novel uses of an IaaS can be evolved by extending existing applications with EbbRT. Libraries of reusable Ebbs that target core primitives such as various types of matrices and associated operations can be developed. The libraries can be used to explore the incremental acceleration of many applications in addition to the wholesale development of new applications. As IaaS providers evolve support for higher performance data-center interconnects with features such as RDMA and native support for collective and reduce operators[6], EbbRT libraries can enable direct application use.

5 Conclusion

We have introduced a new system software runtime called EbbRT. EbbRT explores a unique system architecture, 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] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 95–109, New York, NY, USA, 1991. ACM.

[5] 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.

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

[7] 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.

[8] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing os extensions safely and efficiently with bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 239–252, New York, NY, USA, 2013. ACM.

[9] A. Baumann, M. Peinado, G. Hunt, K. Zmudzinski, C. Rozas, and M. Hoekstra. Secure execution of unmodified applications on an untrusted host. In *Work in Progress - SOSP'13*, 2013.

[10] 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.

[11] M. Ben-Yehuda, O. Peleg, O. A. Ben-Yehuda, I. Smolyar, and D. Tsafrir. The nonkernel: A kernel designed for the cloud. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, pages 4:1–4:7, New York, NY, USA, 2013. ACM.

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

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

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

[16] 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.

[17] Google. V8 javascript engine.

[18] 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.

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

[20] Joyent. Node.js.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] R. Nikolaev and G. Back. Virtuos: An operating system with kernel virtualization. In *Proceedings*

of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, pages 116–132, New York, NY, USA, 2013. ACM.

[30] 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.

[31] N. Piggin. Slqb - and then there were four.

[32] 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.

[33] W. Reese. Nginx: The high-performance web server and reverse proxy. *Linux J.*, 2008(173), Sept. 2008.

[34] 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.

[35] 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.

[36] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[37] 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.

[38] 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.

[39] 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.

[40] D. A. Wheeler. generated using david a. wheeler's 'sloccount'.