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

Overview

- Motivation
- Objectives
- System design
- Implementation
- Evaluation
- Conclusion

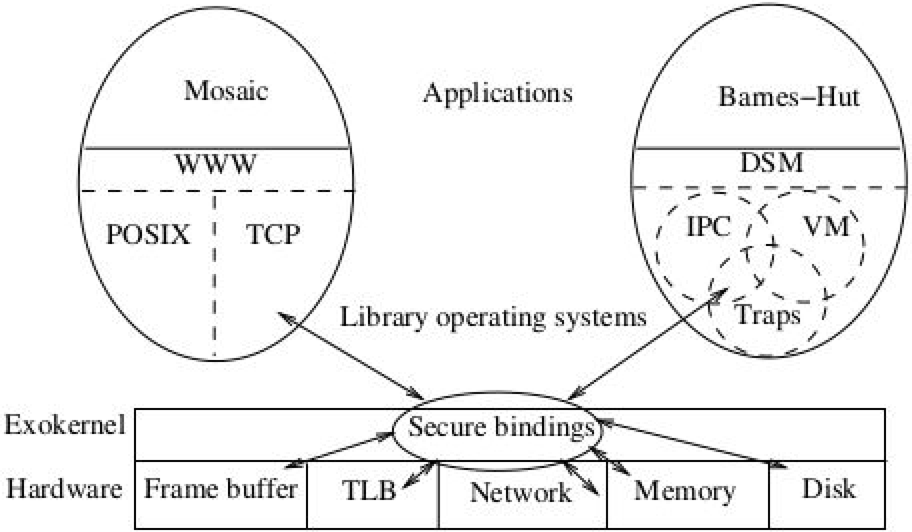
Motivation

- Emphasis on CPU performance and software stack in cloud environments
    - End of Dennard scaling.
    - High speed I/O devices.
- Limitations of generality of commodity operating systems
    - Fixed interface and implementation.
- Techniques in response
    - Hardware virtualization.
    - Kernel bypass techniques.
    - Library operating systems.
- Engineering effort and narrow applicability.

Library OS

- Operating systems define the interface between application and hardware resources.
- They hide information about machine resources behind high level abstractions such as processes, files, address spaces and interprocess communication.
- Certain architectures leave the management of physical resources to applications by exporting hardware resources to library operating systems through low-level interfaces.

Exokernel


Mosaic
Applications
Barnes–Hut
WWW
DSM
POSIX
TCP
IPC
VM
Traps
Library operating systems
Exokernel
Secure bindings
Hardware
Frame buffer
TLB
Network
Memory
Disk

Unikernel

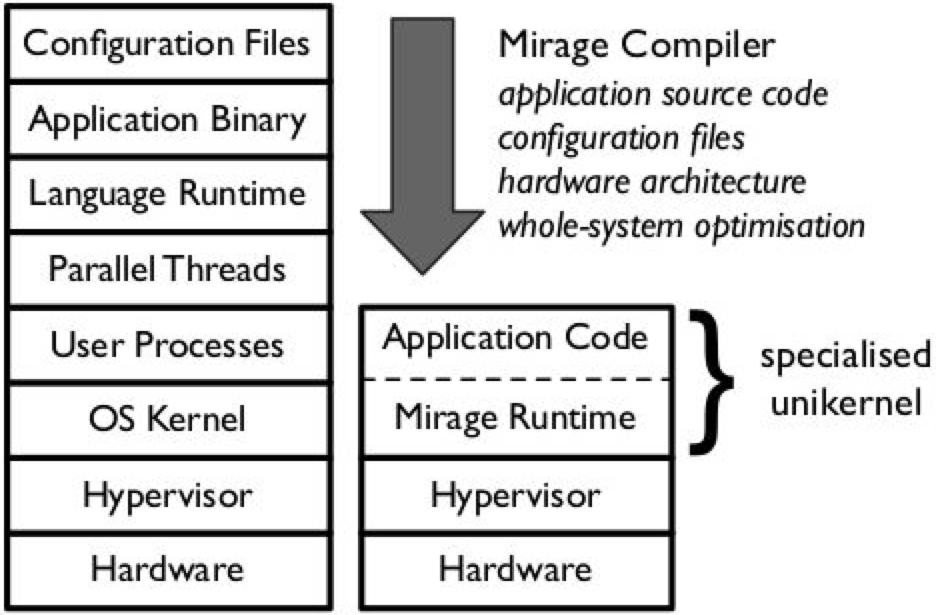


Figure 1: Contrasting software layers in existing VM appliances *vs.* unikernel's standalone kernel compilation approach.

Objectives

- Performance specialization
  - Allow applications to specialize the system at every level.
  - Provide an event driven environment with minimal abstraction over hardware.
  - Low overhead component model to be used throughout performance sensitive paths.
- Broad Applicability
  - Designed to support existing libraries and complex runtimes.
  - Heterogeneous distributed architecture called MultiLibOS model.
  - EbbRT library OS and general purpose OS present.
- Ease of development
  - Exploits modern language techniques to simplify the task of writing software.
  - Ebb model encapsulates existing system components.
  - Difficulty to port applications reduced through function offloading.

System Design

- Heterogeneous distributed structure
- Modular system structure
- Non-preemptive event driven execution model

Heterogeneous distributed structure

- Cloud environment, single application can be deployed across several machines.
- Deployed across a heterogeneous mix of specialized library OS and general purpose OS.
  - Light weight bootable runtime - native runtime.
  - User level library - hosted runtime.
- Native runtime sets up a single address space, basic system functionality (eg. timers, networking, memory allocation) and invokes an application entry point while running at highest privilege level.

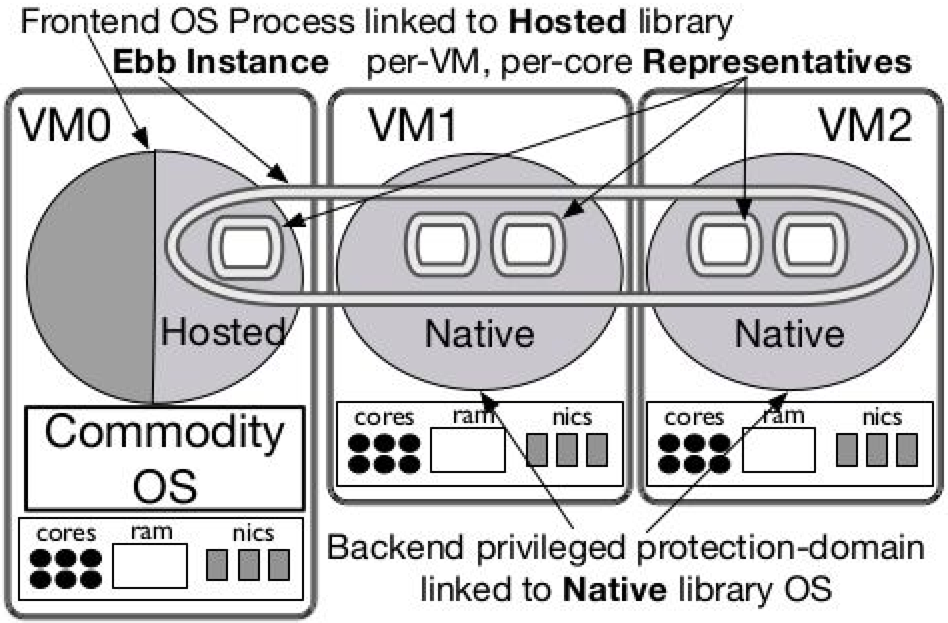


Figure 1: High Level EbbRT architecture

Modular system structure

- Comprised of objects called Elastic Building Blocks.
- Can modify or extend software stack to provide high degree of customization.
- Distributed, multi-core fragmented objects.
- Namespace of Ebbs is shared across both the hosted and native runtimes.

Objects in distributed environment

- Shared objects
- Replicated objects
- Fragmented objects

Adaptable replicated objects

- Replicas enhance availability and reliability in distributed environments.
- Replicas need to be maintained consistent.
- Tradeoff between consistency and performance.
- Consistency contract must be implemented without jeopardizing performance.
- Replica
  - Encapsulates local copy and provides interface to access the object.
- Access object
  - Wrapper that controls accesses to replica.
- Consistency manager
  - Maintains consistency.
- Examples: counter, distributed editor.

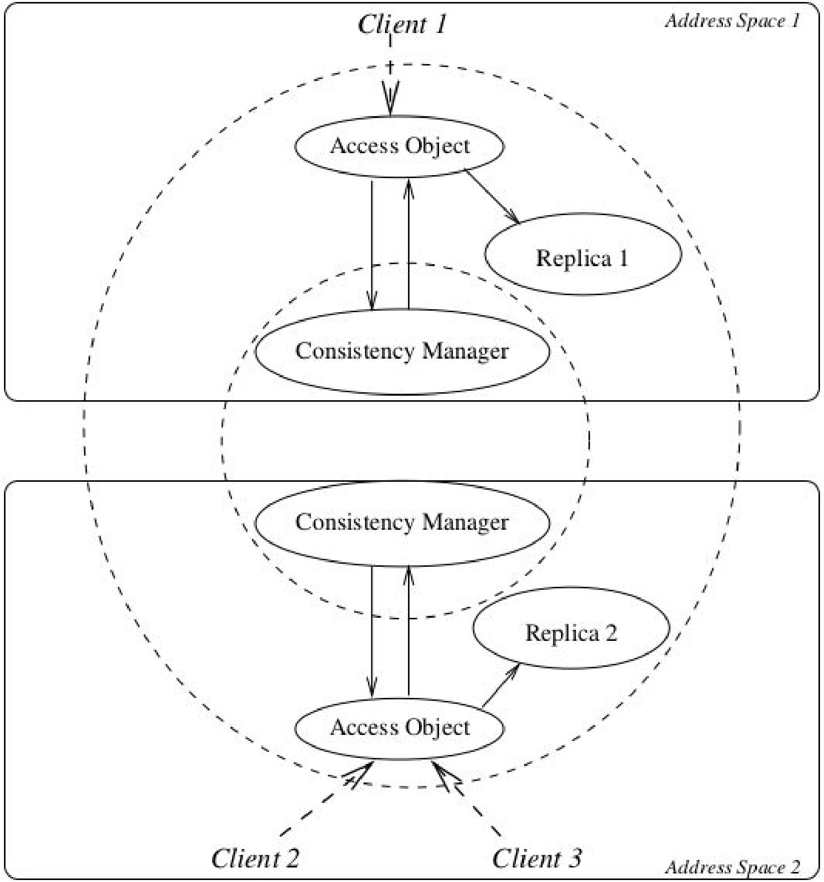


Figure 1: Concrete representation of a replicated object shared by three threads located in two processes. Dashed lines identify logical distributed object boundaries.

Fragmented objects

- A fragmented object (FO) can be viewed at two different levels of abstraction
  - Client’s view (external/abstract).
  - Designer’s view (internal/concrete).
- For clients, FO is a single shared object.
- For designers, FO is composed of
  - Set of elementary objects, *fragments*.
  - Client interface exported through *public interface*.
  - Interface between fragments, *group interface*.
  - Lower level shared FOs used for communication, *connective objects*.

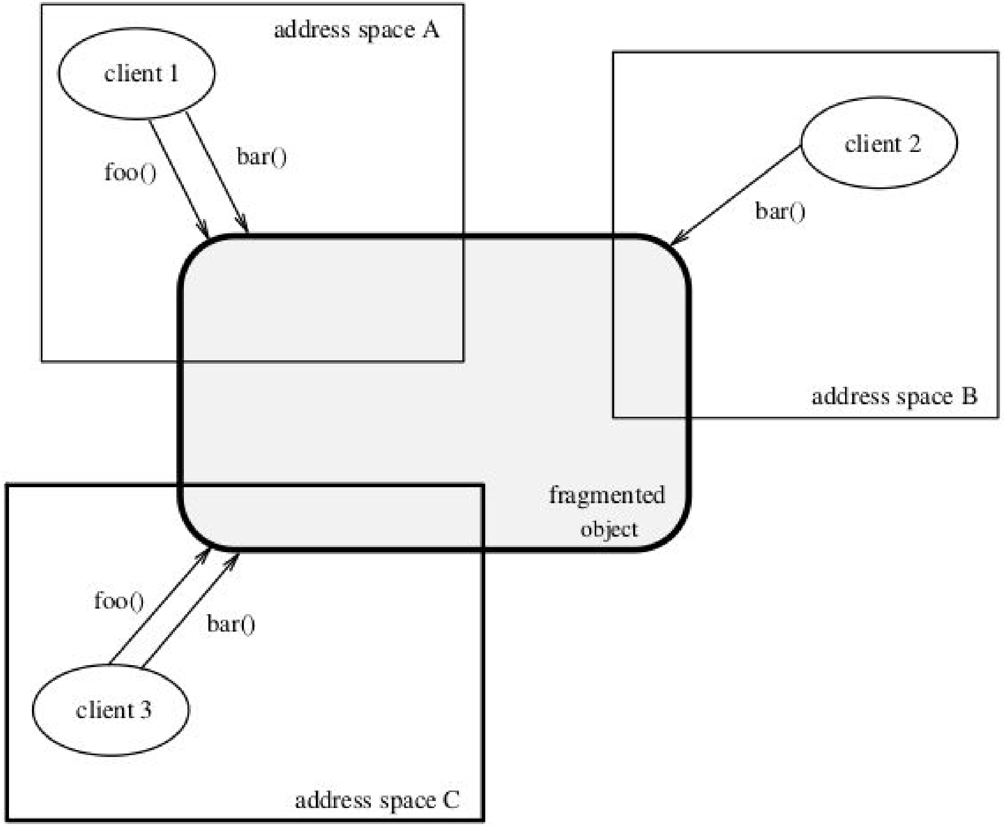


Figure 1: A fragmented object as seen from clients

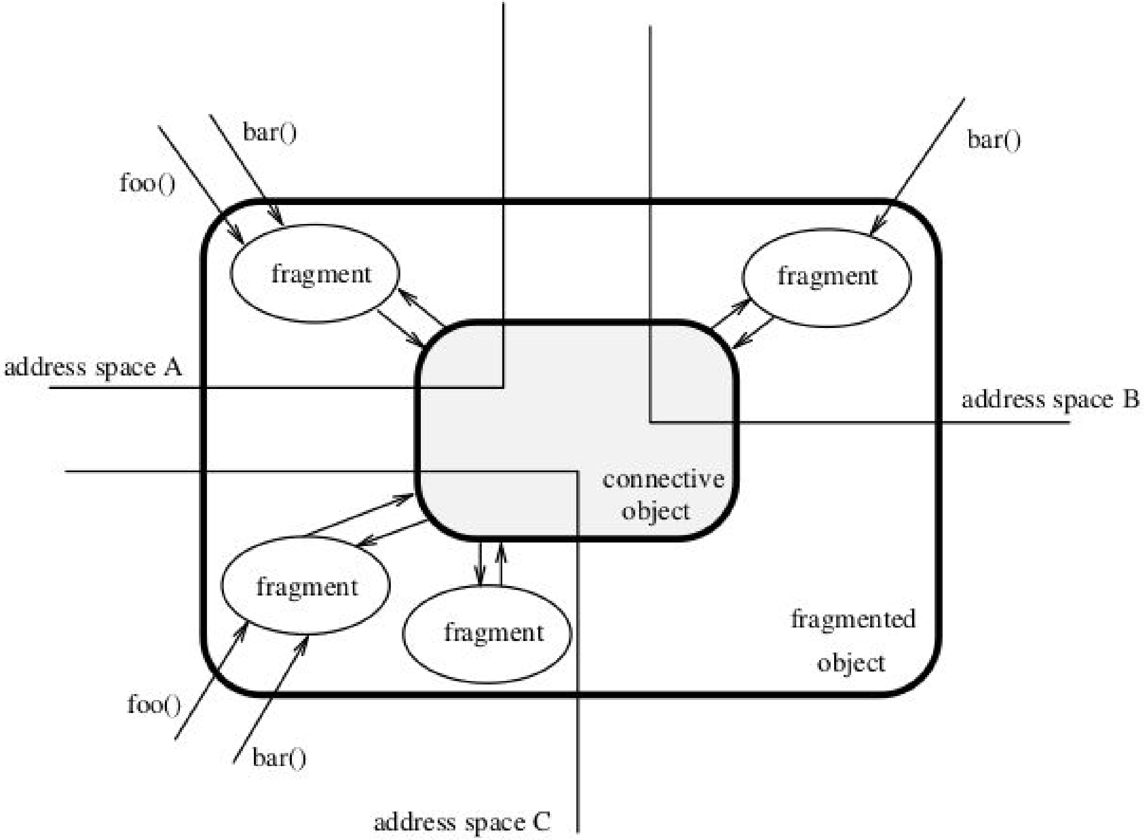


Figure 2: A fragmented object as seen by its designer

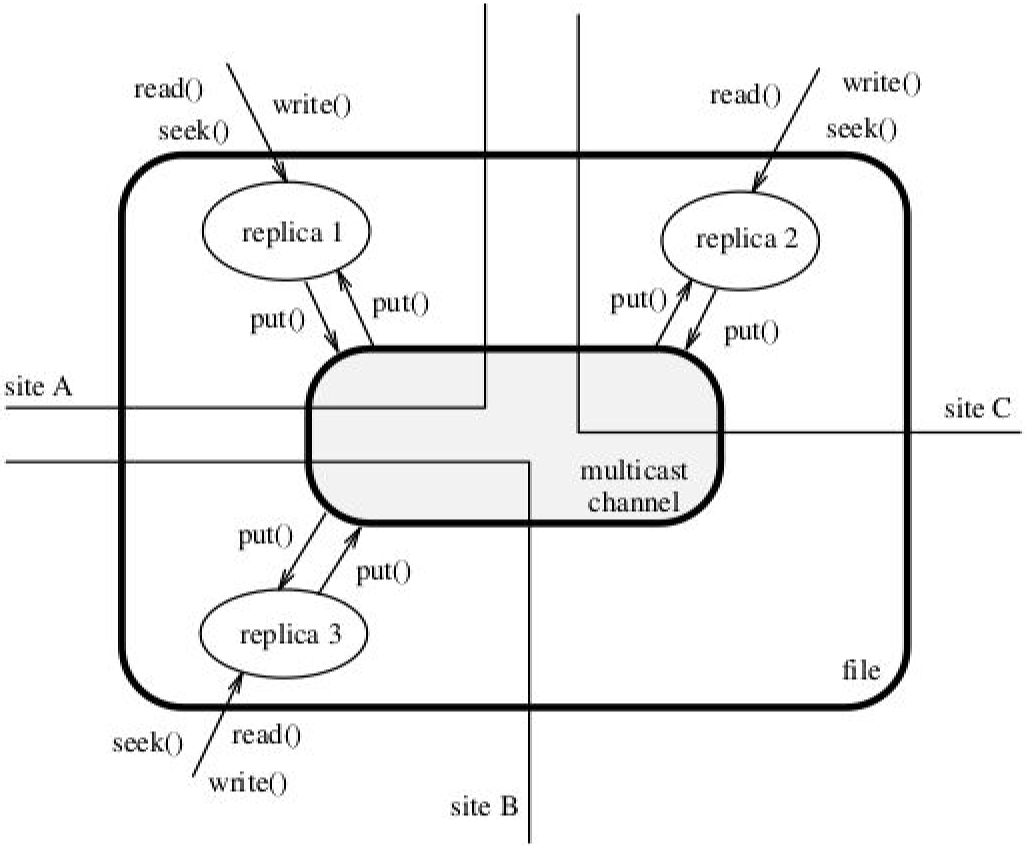
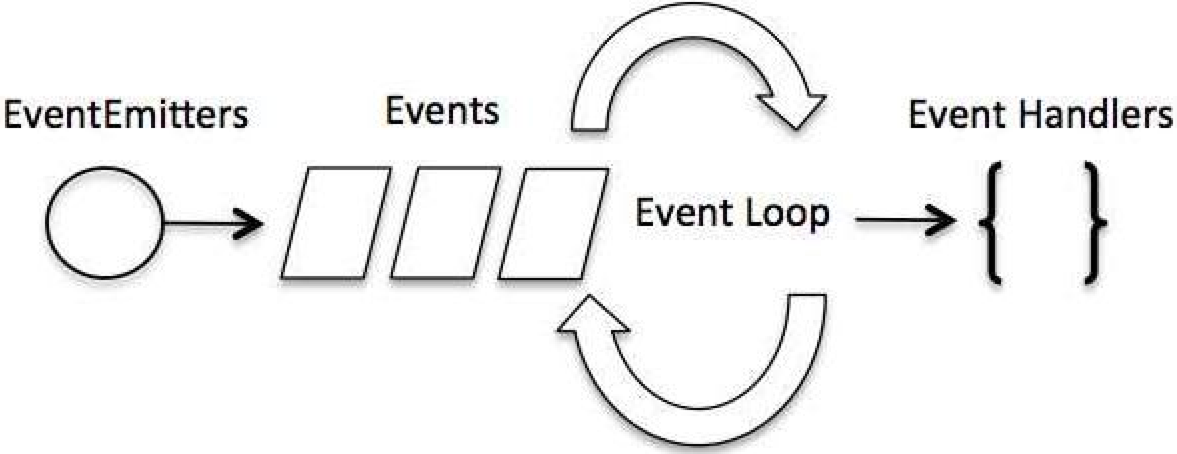


Figure 3: The fragmented representation of a replicated file

Execution model

- Non-preemptive and event-driven.
- Event loop per core
  - Dispatches external and software generated events to registered handlers.
- Hosted library provides analogous environment through the use of poll or select.
- Cloud applications driven by external requests in general
  - Event driven programming a natural choice.
- Cooperative threading model provided as well
  - Blocking semantics and concurrency model similar to Go.

Event driven execution



Implementation

- Software system overview
- Events
- Elastic Building Blocks
- Memory management
- Lambdas and futures
- Network stack

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

Table 1: The core Ebbs that make up EbbRT. A gray row indicates that the Ebb has a multi-core implementation (one representative per core) while the others use a single shared representative.

Software system overview

- Written predominantly in C++14.
- Native library is packaged with GNU toolchain and libc modified to support x86_64-ebbrt build target.
- Application when compiled with toolchain produces a bootable ELF binary linked with library OS.
- POSIX incompatible. Too restrictive and unnecessary.
- Provides necessary functionality for events to execute and Ebbs to be constructed and used.

Events

- Both native and hosted systems provide event driven execution
  - Uses Boost ASIO library to interface with system APIs.
  - Event driven API implemented directly on hardware.
- Drivers allocate an interrupt from Event manager and bind a handler.
- Execution begins at the top frame of a per-core stack.
- Exception handler checks for event handler bound to interrupt and invokes.
- Events typically generated by hardware interrupts.

Synthetic Events

- Can invoke synthetic events on any core in the system.
- Spawn method
  - Receives an event handler that is later invoked.
  - Executed only once.
- IdleHandler
  - Handler for recurring events.

Event Manager

- Priority Order
  - Handles any pending interrupts.
  - Dispatches a single synthetic event.
  - Invokes all idle handlers.
  - Enables interrupts and halts.
- Adaptive polling implementation
  - Device programmed to fire interrupt when packets are received.
  - Process each packet to completion.
  - Rate beyond a threshold install IdleHandlers instead to poll the device.

Limitations

- Cooperative threading model.
- Long running threads
  - Preemptive scheduler.
  - Dedicated processors.
  - Cloud applications IO driven.

Elastic Building Blocks

- Nearly all software in EbbRT is written as elastic building blocks.
- Every instance is identified by a system wide unique EbbId.
- EbbId provides an offset into a virtual memory region backed with distinct per-core pages which holds a pointer to the per-core representative.
- When function is called and the pointer is null a type specific handler is invoked which either returns a reference to a representative or throws a language level exception.
- Fault handler will construct and store the representative so future invocations take the fast path.
- Hosted implementation uses per-core hash tables.

- EbbRT provides core Ebbs that support distributed data storage and messaging services.
- Fast path cost of a Ebb invocation is one predictable branch and one unconditional branch more than a normal C++ object dereference.
- Avoided using interface definition languages.

Memory Management

- Similar to that of Linux Kernel.
- Page Allocator
  - Buddy allocator per NUMA node.
- Slab Allocator Ebbs
  - Allocate fixed sized objects.
  - Per core, per NUMA node representatives to store free lists and partial pages.
  - Design based on Linux Kernel's SLQB allocator.
- General Purpose Allocator
  - Slab Allocator.
  - VMem Allocator.

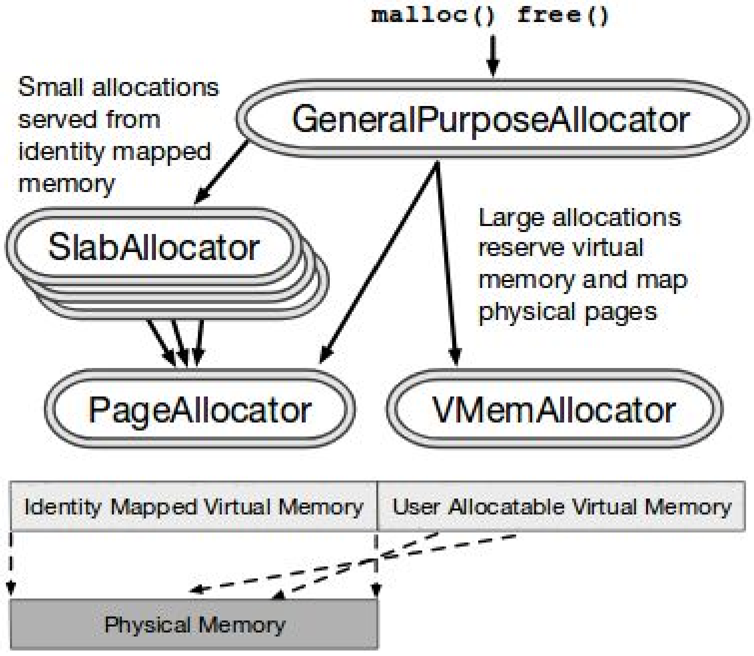
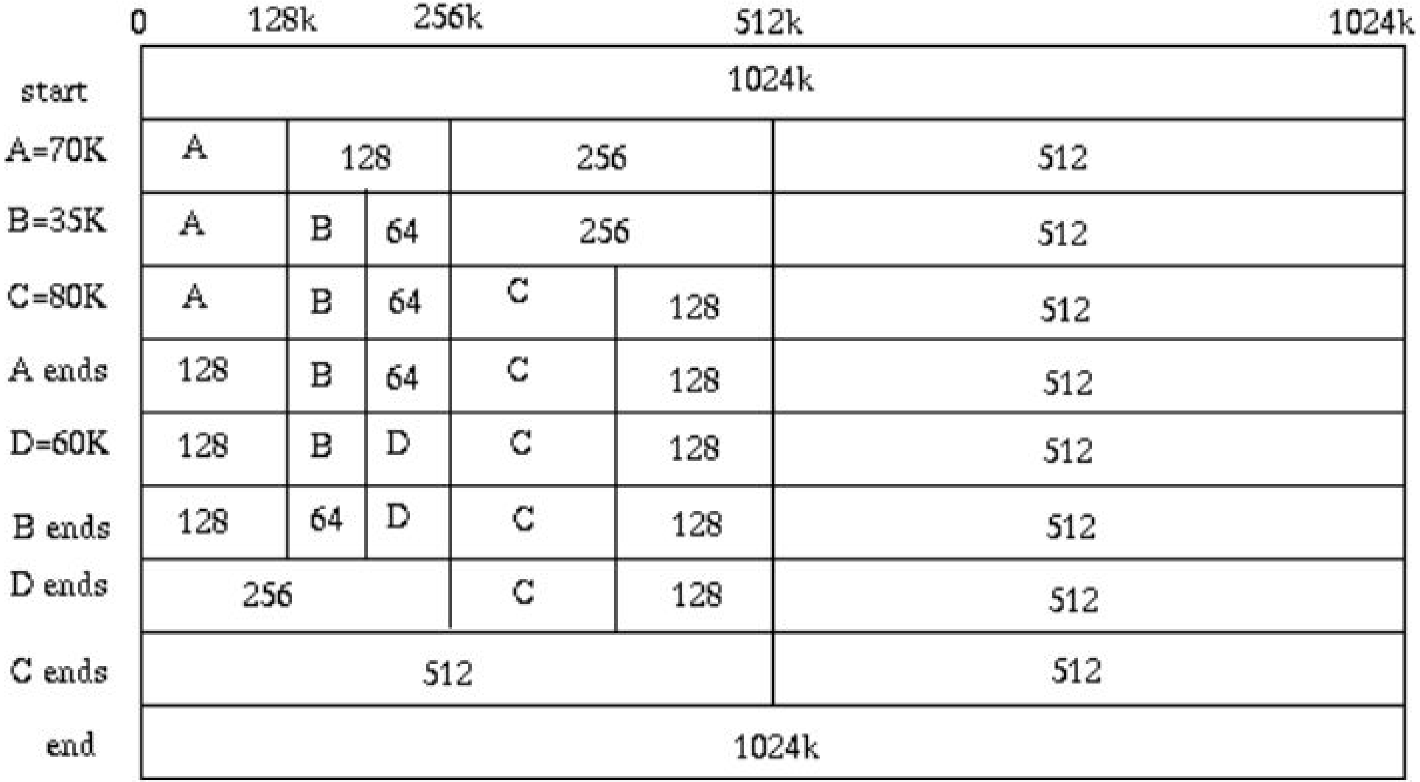


Figure 2: Memory Management Ebbs

Buddy Allocator



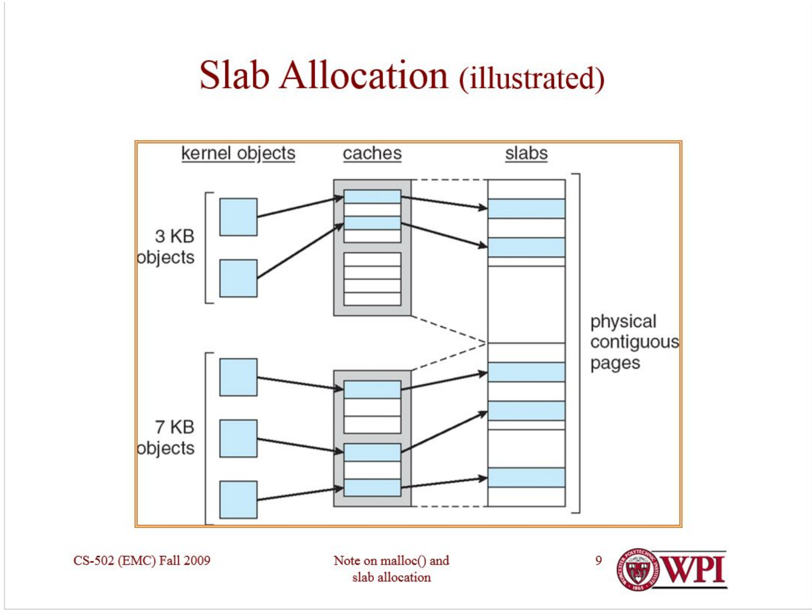
	0	128k		256k		512k	1024k
start	1024k						
A=70K	A	128		256		512	
B=35K	A	B	64	256		512	
C=80K	A	B	64	C	128	512	
A ends	128	B	64	C	128	512	
D=60K	128	B	D	C	128	512	
B ends	128	64	D	C	128	512	
D ends	256			C	128	512	
C ends	512					512	
end	1024k						

Buddy Allocator

- Generally implemented using binary trees.
- Very little external fragmentation.
- Low compaction overhead
- Problem - internal fragmentation due to memory wastage

Slab Allocator

- Each page - only to a particular type of object.
- Free lists maintained for each of the partial slabs.
- Advantages :
    - No external fragmentation
    - Data structures of some objects can be difficult to move than other objects. So paging policies can be changed to include for this fact.



Slab Allocation (illustrated)

CS-502 (EMC) Fall 2009

Note on malloc() and slab allocation

9

WPI

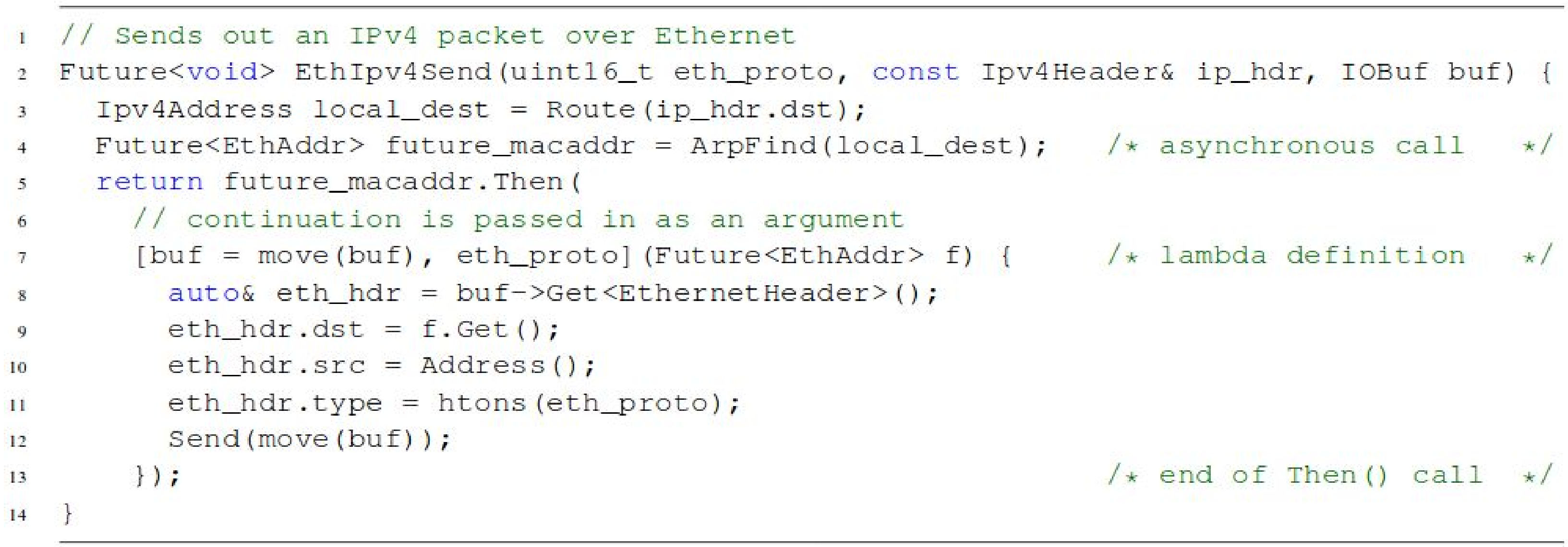
- Any ebb can be modified/replaced without impacting others.
- Compiler optimization, function inlining.
- Can perform zero copy IO, when memory is identity mapped rather than allocating memory for DMA.
- Lack of preemption
  - Allocations served from per core cache without synchronization.
- Partition of virtual memory.
- VMem Allocator allows implementation of arbitrary paging policies.

Advantages

- Scalability: per core representatives.
- Lack of preemption: no need for synchronization.
- Library OS design: tighter collaboration between system and application components.
    - Directly manage virtual memory
    - Achieve zero copy interactions with device.

Event driven programming limitations

- Obfuscates control flow of application
    - Example: asynchronous calls, construct continuations - control mechanisms to save and restore state across invocations.
    - Lambdas capture local state that can be referred when they are invoked.
- Complex error handling
    - Exceptions in c++.
    - Stack unwound to most recent try catch block.
    - One logical flow of control split across multiple stacks.
    - Exceptions must be handled at every event boundary.
    - Monadic futures used instead.



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

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

- Futures - datatype for asynchronously produced values
- A future cannot be directly operated on, instead lambda can be applied using THEN method.
- Lambda is invoked once the future is fulfilled.

- THEN function returns Future representing value returned by applied function.
- This allows other software components to chain further functions to be invoked on completion.
- Any exception will flow to the first function which attempts to catch the exception - behaviour similar to synchronous code.
- C++ futures have no THEN function, block then using get function.
- Futures - interface definitions, lambdas - manual continuation construction

Network Stack

- Did not port but implemented the network stack anew.
- Features: IPv4, TCP/IP, DHCP functionality
    - Provided event driven interface to applications.
    - Minimized multi-core synchronization.
    - Enabled pervasive zero copy.
- Does not provide standard BSD socket interface.
- Enables tighter integration with application to manage resources.

- IOBuf primitive to support zero-copy software.
- Manages ownership of a region of memory as well as view of a part of it.
- Applications do not invoke read on a buffer.
- Rather they install a handler which is passed an IOBuf.
- Network stack does not provide buffering but will invoke the application as long as data arrives.

- Most systems have fixed size buffers to pace connections.
- Application can manage its own buffering.
- UDP drop datagrams.
- TCP set window size to prevent further sends.
- Check if outgoing data fits within the advertised window.
  - If yes send otherwise buffer.
- Allow applications whether to delay sending to aggregate multiple sends.
  - Other Systems - Nagle’s algorithm - poor latency.
  - EbbRT - applications can tune behaviour of it’s connections runtime
- Default behaviours provided.

- Challenge - Synchronizing accesses to connection state.
- Connection state is stored in a RCU hash table.
    - No atomic operations required.
- Connection state manipulated only by a single core, chosen by application.
- Common case network operations require no synchronization.
- Network stack specialization
    - Buffering and queuing important factor in performance.
    - EbbRT gives more control to the applications
    - Zero copy optimization illustrates the value of having physical memory identity map, unpaged and within single address space.

Evaluation

- Affirm that this fulfills all the three objectives discussed.
  - Supports High-performance specialization
  - Provides support for broad set of applications
  - Simplifies development of application-specific systems software
- Micro-benchmarks to quantify base overheads of primitives.
- Macro-benchmarks that exercise EbbRT in the context of real applications.

Microbenchmarks

- Evaluates memory allocator and overheads of Ebb mechanism.
- Evaluates latencies and throughput of network stack and exercise several of system features discussed including idle event processing, lambdas and IOBuf mechanism.

Memory Allocation

- Ported Threadtest from Hoard benchmark suit.
- Compared performance with glibc 2.2.5 and jemalloc 4.2.1 allocators.
- Allocator scales competitively with production allocators.
- Scalability due to locality induced by the per-core Ebb reps of mem allocator and lack of preemption which removes synchronization.

Memory Allocator

- Each thread T allocates N * 8 / T byte objects.

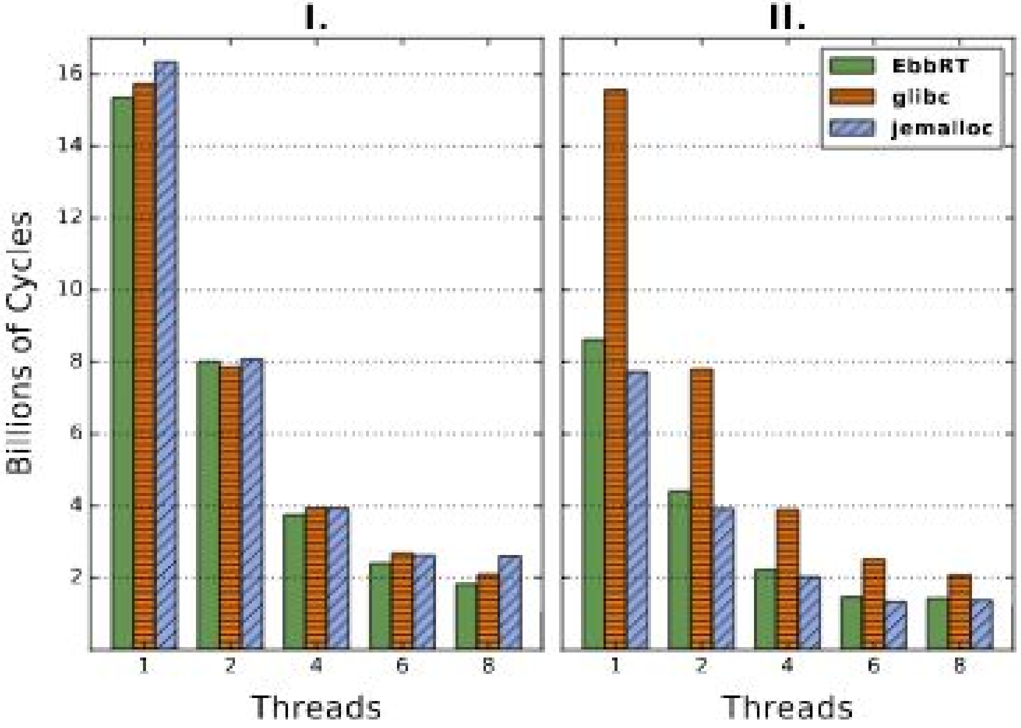


Figure 4: Hoard Threadtest. Y-axis represents threads, t. I.) N=100,000, i=1000; II.) N=100, i=1,000,000.

Network Stack

- Ported NetPIPE and iPerf benchmarks.
- NetPIPE
    - Client sends a fixed size message to server which is echoed back after receiving it completely.
    - Illustrates latency of sending and receiving over TCP.
- iPerf
    - Client opens a TCP stream and sends fixed sized messages which server receives and discards.
    - Confirms - run-to-completion network stack does not preclude high throughput applications.
- EbbRT servers - 24.53 microsec, 64 B msg - 4Gb goodput, 100 kB
- Linux VMs - 34.27 microsec, 64 B mgs - 4GB goodput, 200 kB
- EbbRT short path achieves a 40% improvement in latency.
- This illustrates the benefits of non-preemptive event driven execution model and zero copy instruction path.

Network Stack

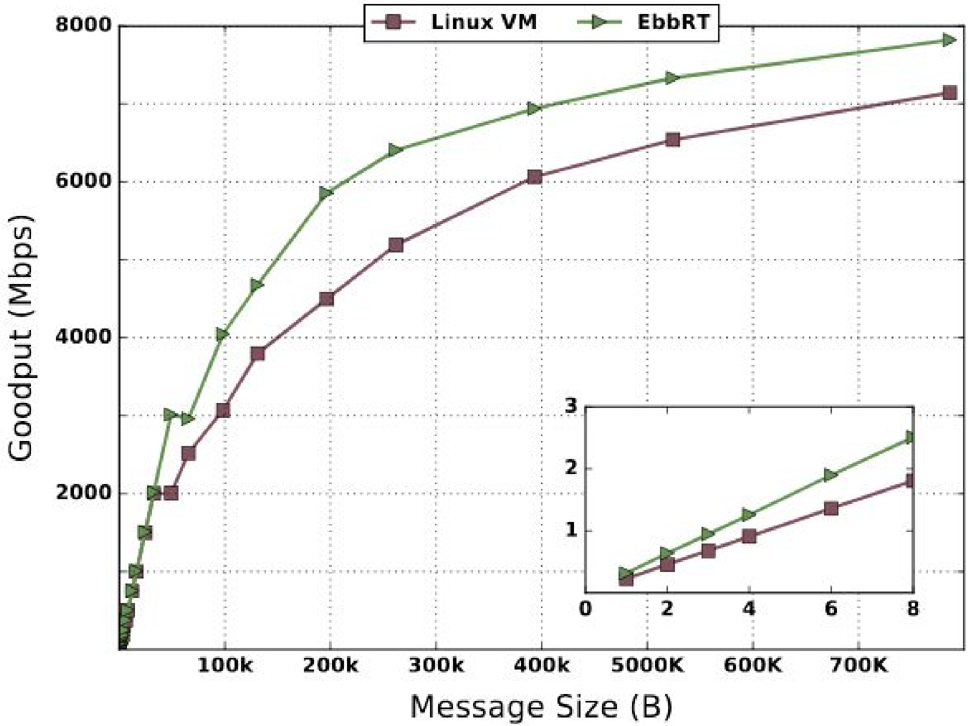


Figure 5: NetPIPE performance as a function of message size. Inset shows small message sizes.

Memcached

- Distributed memory caching system - caches data and objects in RAM to reduce number of times an external data source must be read.
- Used mainly in dynamic web applications to reduce database load.
- In memory key-value store common benchmark in examination and optimization of networked systems.
- Significant OS overhead for Memcached
- Re-implemented Memcached instead of porting.
- Supports standard memcached library protocol.
- Key value pairs stored in RCU hash table to alleviate lock contention.

Memcached

- Benchmarking tool - Mutilate
- Place particular load on server and measure response latency.
- Configure to generate load representative of facebook ETC workload.
  - Consists 20-70 B keys and 1-1024 B values.

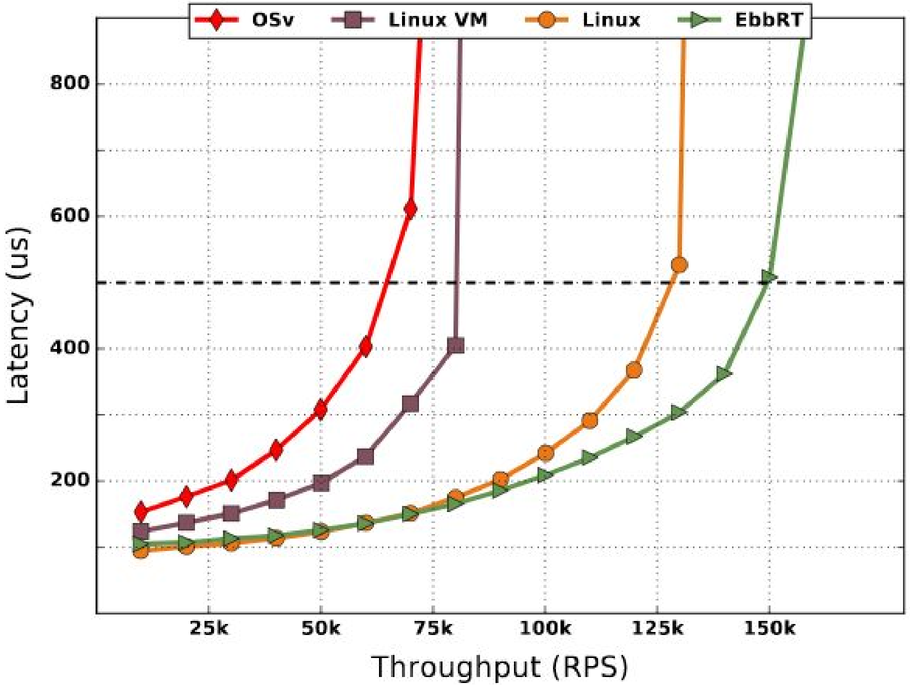


Figure 6: Memcached Single Core Performance

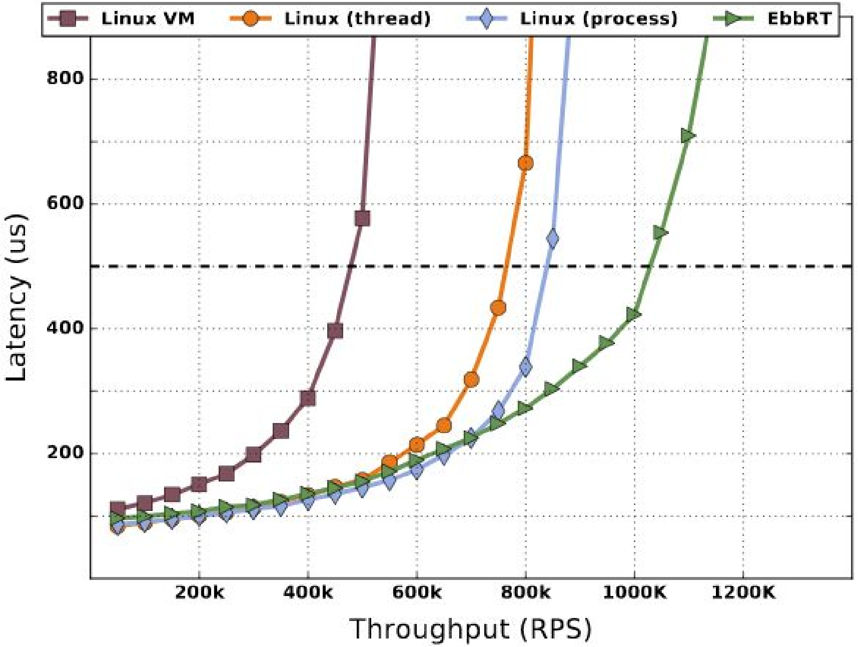
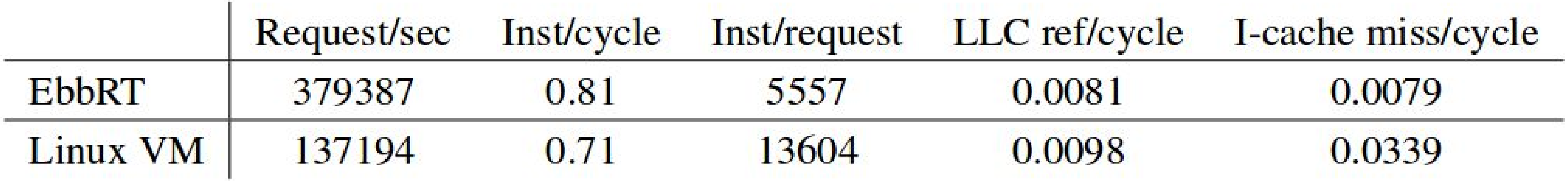


Figure 7: Memcached Multicore Performance

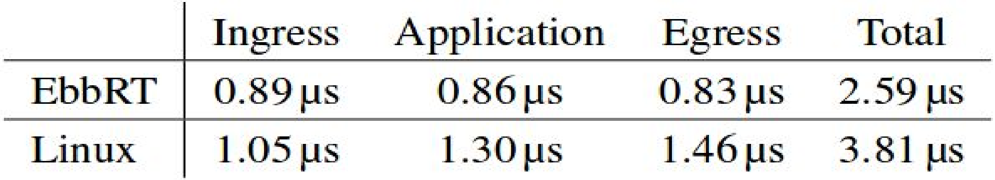
Represents 99th percent latency



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

Table 2: Memcached CPU-efficiency metrics

- Linux Kernel perf utility used to gather data - 10 sec duration of a fully-loaded single core memcached server run within a VM
- 2.75x speedup for request processing - shorter non-preemptive instruction path for processing requests.

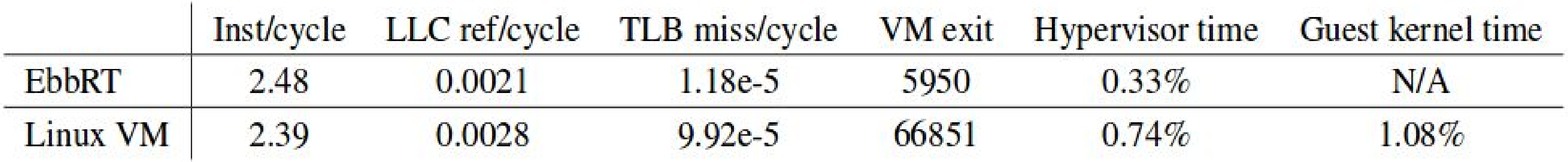


	Ingress	Application	Egress	Total
EbbRT	0.89 µs	0.86 µs	0.83 µs	2.59 µs
Linux	1.05 µs	1.30 µs	1.46 µs	3.81 µs

Table 3: Memcached Per-Request Latency

Node.js

- In comparison to memcached node.js uses many features like virtual memory mapping, file I/O, periodic timers etc.
- To illustrate EbbRT's support for broad class of software, also reducing developers burden required to develop specialized systems.
- Benchmark - V8 Javascript benchmark suite



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

Table 4: V8 JavaScript Benchmark CPU-efficiency metrics

Score - inversion of running time, scaling by the score of a reference implementation, geometric mean of 8 scores

Inefficiency of Linux VM - executes more instructions such as VM exits, extraneous Kernel functionality like scheduling etc.

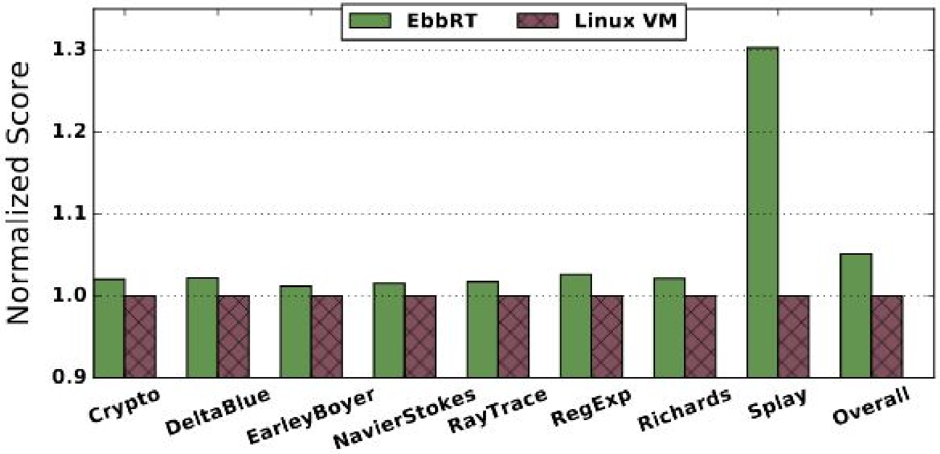


Figure 8: V8 JavaScript Benchmark

Node.js Webserver

- WRK benchmark - place moderate load on the webserver.
- EbbRT - 91.1µs mean and 100µs 99th percentile latencies.
- Linux - 103.5µs mean and 120.6µs 99th percentile latencies
- Linux has 13.6% higher mean latency and 20.65% higher 99th percentile latencies over EbbRT.

Conclusion

- Library OS uses - portability, security, efficiency
- EbbRT applications achieve high performance through system wide specialization rather than one particular technique.
- Long-term goal - ability to be used for a broad range of applications, enabling high degree of specialization
- EbbRT framework for constructing specialized systems for cloud applications