PIKA: A Network Service for Multikernel Operating Systems

Nathan Z. Beckmann, Charles Gruenwald III, Christopher R. Johnson, Harshad Kasture, Filippo Sironi
Anant Agarwal, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

ABSTRACT

PIKA is a network stack designed for multikernel operating systems that target potential future architectures lacking cache-coherent shared memory but supporting message passing. PIKA splits the network stack into several servers that communicate using a low-overhead message passing layer. A key challenge faced by PIKA is the maintenance of shared state, such as a single accept queue and load balance information. PIKA addresses this challenge using a speculative 3-way handshake for connection acceptance, and a new distributed load balancing scheme for spreading connections. A PIKA prototype achieves competitive performance, excellent scalability, and low service times under load imbalance on commodity hardware. Finally, we demonstrate that splitting network stack processing by function across separate cores is a net loss on commodity hardware, and we describe conditions under which it may be advantageous.

1 INTRODUCTION

Recent research has proposed several distributed kernel (i.e., multikernel) architectures for multicore processor operating systems [6, 24]. These multikernel architectures do not share data structures among cores to avoid scalability bottlenecks that have plagued monolithic kernels [7]. Multikernels are particularly suitable for multicore processors that do not support cache-coherent shared memory [15]. Although the jury is out whether future multicore and manycore processors will support cache-coherent shared memory or not [17], an interesting research question to explore is how to design and implement system services that require shared state but cannot rely on cache-coherent shared memory. This paper explores the design and implementation of one such system service, the network stack, and proposes a novel design that achieves good performance and scalability under a wide range of workloads.

A simple design for a multikernel network stack is to have one instance on one core shared between all applications (e.g., web servers) [20], but this design does not scale well with an increasing number of cores running application code. The other extreme is to have a replica of the network stack on several cores without sharing, but this design may result in low utilization and makes it challenging to share, for instance, a single TCP port for incoming connections. This paper investigates a split design for the network stack [6, 24], called PIKA, consisting of a collection of servers (i.e., user-space processes) that collaborate to provide high-performance, scalable networking. PIKA's instance of a split design has one or more Link Servers that manage the network interface cards (NICs), one or more Transport Servers that provide transport services such as TCP, and one or more Connection Managers that are in charge of connection establishment. Our implementation allows each of these components to either be run as separate servers each on their own core or to be combined into hybrid servers on fewer cores. We evaluate the performance implications of each of these design alternatives and the conditions under which splitting network stack processing may be advantageous.

The biggest challenge in PIKA's design is the management of shared state among the servers without the convenience of cache-coherent shared memory. Specifically, connection management requires the abstraction of a single accept queue to distribute connections among applications.[1] Additionally, maintaining high utilization in the presence of applications with high and low service time requires load balancing of connections among them. The challenges of shared state are most apparent in short-lived connections, which stress the accept queues and load balancer. PIKA addresses these challenges with a speculative 3-way handshake protocol to accept connections and a novel distributed load balancing scheme.

Moreover, as computation has moved into the data center and multicore processors have encouraged parallel computing, response times have come to be dominated by the long tail of service time distributions [5, 9]. It is increasingly important to design network services not only for throughput, but for worst-case latency. Load balancing is thus a first-order challenge for PIKA. Our evaluation shows that PIKA maintains low service times across a variety of load imbalances.

The main contributions of this paper are as follows. First, we show that a network stack for a multikernel can achieve good performance and scalability without reliance on cache-coherent shared memory. Second, we present a novel load balancing scheme that allows the network stack to achieve high throughput and low latency in the presence of application delays. Third, we evaluate various design choices for a network stack on a multikernel via a thin message passing layer built on top of the Linux kernel to emulate a multikernel architecture. Using this setup we investigate which

[1]Throughout this paper, we use *application* to mean a single instance of an application (i.e., a process).

design performs best on commodity hardware. Combined, these contributions serve as a template for the design and implementation of network services for multikernels.

The rest of the paper is organized as follows: Section 2 provides an overview of related work. Section 3 discusses the design of the system and the various components of the networking service. Section 4 provides implementation details while Section 5 presents experimental results. Finally, Section 6 concludes the paper.

2 Related work

PIKA is the first network stack design we are aware of that achieves good performance, scalability, and load balance without reliance on cache-coherent shared memory. In particular, PIKA tackles challenges similar to those addressed in previous research on multikernels and microkernels with split network stack designs, load balancing of connections, new application programming interfaces (APIs) for monolithic kernels, and techniques to avoid scaling problems due to data sharing (e.g., data structures and locks) in monolithic kernels.

Split Network Stack Designs. PIKA adopts a split network stack design from the Barrelfish [6] and fos [24] multikernels. PIKA extends the design with support for load balancing to distribute connections across the different applications, providing fast emulation of a shared accept queue. Although our evaluation uses a modern 10 Gigabit Ethernet (GbE) NIC instead of a 1 GbE NIC (yielding $10\times$ higher peak throughput), we achieve approximately $100\times$ the throughput of Barrelfish and fos. PIKA achieves these gains despite the fact that higher throughput stresses PIKA's design much more than previous research. Our evaluation further explores the best configuration of components on commodity hardware and how architectural changes could affect this decision.

Hruby et al. [16] also proposed splitting the network stack into several user-space processes that communicate via message passing. However, they focused primarily on reliability, and their design factors the network stack into components that manage different layers and protocols (e.g., network – IP, transport – TCP and UDP, etc.) to provide fault isolation. PIKA, by contrast, is focused on performance and scalability by minimizing the shared state among components.

Load Balancing of Connections. PIKA employs a dynamic, distributed load balancing algorithm to distribute connections to applications: it determines how to distribute connections based on the current state of the system, in a decentralized fashion (e.g. it does not rely on a centralized server for making load balancing decisions[10, 21]).

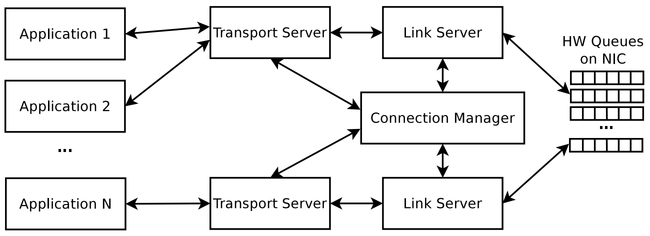
It is assumed that connections accepted by an application under PIKA are processed by that same application until completion, so there is no load balancing of connections between applications. Our load balancing algorithm is therefore similar to web server load balancing which distributes connections across several web server instances capable of satisfying the request [8]. In those systems, the decision of which web server receives a connection is left to the client, a DNS server, or a separate dispatcher system which may take factors such as server load and network congestion into account [13]. PIKA's load balancer, by contrast, is integrated with the network stack on the system hosting the application rather than as a discrete component and therefore does not need to take outside network state into account.

New APIs for Monolithic Kernels. A number of researchers have proposed changes to the interfaces of monolithic kernels to address scaling problems. Soares and Stumm [22] proposed FlexSC, an exception-less system call mechanism for applications to request system services. FlexSC batches synchronous system calls via asynchronous channels (i.e., system call pages); system call kernel threads running on dedicated cores provide system services, thus avoiding trapping from user to kernel mode and addressing cache pollution problems. The synchronous execution model is preserved by a user-space threading library that is binary-compatible with the POSIX threading API and also makes FlexSC transparent to legacy applications, while the asynchronous execution model is supported by libflexsc [23]. PIKA also dedicates multiple cores to network stack processing but assumes hardware supporting message passing instead of cache-coherent shared memory.

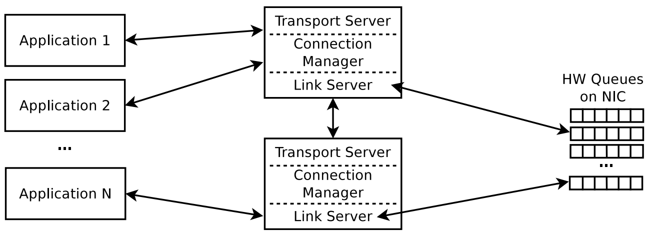
Han et al. [12] presented MegaPipe, a new API for scalable network input/output (I/O). MegaPipe employs thread-local accept queues, which are responsible for connection establishment on a particular subset of cores, thus avoiding remote cache accesses. MegaPipe also batches asynchronous I/O requests and responses via synchronous channels with traditional exception-based system calls to favor data cache locality. PIKA attains good performance and scalability implementing the POSIX socket API, thus supporting unmodified applications. Moreover, PIKA achieves good load balance for both uniform and skewed workload thanks to connection load balancing, which is not supported by MegaPipe.

Avoiding Scaling Problems in Monolithic Kernels. Recent work by Boyd-Wickizer et al. [7] and Pesterev et al. [18] has focused on fixing scaling problems in the Linux kernel's network stack due to data sharing through cache-coherent shared memory. In contrast, PIKA assumes hardware lacking cache-coherent shared memory and solves these problems by other means; hence, these techniques are not applicable. This paper demonstrates that the benefits of Affinity-Accept [18] are attainable in a multikernel design on hardware lacking cache-coherent shared memory but supporting message passing.

Shalev et al. [20] describe IsoStack, a network stack design that eliminates unnecessary data sharing by offloading network stack processing to a dedicated core as in the multikernel architecture [6, 24]. A prototype built inside the AIX kernel with support from a user-space library exploits message queues, event-driven operation, and batching to achieve high



(a) Split system architecture.



(b) Combined system architecture.

Figure 1: Split and combined system architectures. In the split configuration (a) the components are divided apart and run separately. The number of instances of each component is configurable. For the combined configuration (b) the key consideration is the number of instances of the network stack to run.

data throughput matching line speed. PIKA harnesses similar techniques but exploits multiple dedicated cores exploring various parallel configurations to achieve both high connection throughput and low response latency under uniform and skewed workloads. With the same number of concurrent connections, PIKA achieves comparable performance at low core counts and up to an order of magnitude higher performance at high core counts, which IsoStack cannot exploit.

3 DESIGN

The goal for PIKA is to provide a high-performance, scalable network stack for a multikernel across a wide range of workloads. The main challenge is managing shared state to implement POSIX semantics without reliance on cache-coherent shared memory. PIKA load balances connections among servers to maximize throughput and minimize latency. Since multikernels cannot rely on cache-coherent shared memory to share state, PIKA uses message passing to share information and employs a number of techniques to minimize the number of messages sent while maintaining correctness and achieving good performance. This section describes elements of PIKA's design and the techniques it uses to achieve its goals.

3.1 PIKA Components and Design Choices

Like FlexSC [22], PIKA employs dedicated cores for network stack processing which are distinct from application cores. Furthermore, PIKA splits the network stack into various conceptual components based on functionality. The design of PIKA allows each of these components to either be run as stand-alone servers or combined with other components into a composite server. An instance of a PIKA system consists of one or more servers, each of which may encapsulate one or more of these components (Figures 1a and 1b).

The Link Server (LS) is the component that interacts directly with the NIC. Its responsibilities include configuring the NIC (e.g., managing the flow director), sending packets on the hardware interface in response to requests from other components of PIKA, as well as to receive inbound packets and transfer them to the appropriate PIKA component. In particular, it does packet inspection to forward `SYN` packets to Connection Managers (CMs) as described below. The number of LSs desired depends on the hardware configuration. For instance, it may not be desirable to have more LSs than the number of hardware direct memory access (DMA) rings that the NIC can support, since having multiple LSs share a single ring may affect scalability.

The Transport Server (TS) component is responsible for managing TCP/IP flows, including packet encapsulation/de-encapsulation, Transmission Control Block (TCB) state management, out-of-order packet processing and re-transmissions. There is no shared state among separate flows, and thus different flows can be managed by different TSs entirely in parallel. The number of TSs can thus be scaled up trivially to meet the system requirements. Note also that other network stack protocols such as UDP, ICMP, DHCP, DNS, and ARP are also handled by the TS. However since these protocols either do not demand high throughput or are stateless they are not considered in any further detail in this paper.

The CM is responsible for TCP/IP connection establishment. The CM encapsulates all the shared state necessary for the implementation of TCP/IP sockets that adhere to the POSIX socket specification. In particular, it maintains the listen queues where incoming connection requests are enqueued and then distributed to listening applications. The CM also decides which TS should manage each established connection. The CM shares state with the application (Subsection 3.2), and also with other CMs in order to correctly implement POSIX sharing semantics (i.e., a single listen socket shared among multiple application processes) and to effectively load balance among various application processes listening on the same port (Subsection 3.3). Scaling up the number of CMs has associated challenges and trade-offs. A single CM for the entire system can become a bottleneck when high connection throughput is required. On the other hand, deploying multiple CMs presents challenges in sharing state to ensure correctness and high performance (discussed in Subsection 3.2 and Subsection 3.3), since the overhead of maintaining shared state between CMs increases as the number of CMs grows.

In addition to the parallelism within each component, another design consideration is when to combine various components into a single composite server. Combining components (Figure 1b) has the benefit of using fewer cores and reducing context switch overhead over a split configuration (Figure 1a). Additionally, communication between components within a composite server has much lower overhead (function calls vs. messages). This unfortunately leads to loss of control over the level of parallelism within each component. For example, one might want to have many TSs in the system, but not as many CMs (because of the high cost of sharing) or LSs (if the NIC only has a few hardware DMA rings). Splitting components into stand-alone servers may have advantageous cache effects (eliminating cache conflicts between components and reducing the working set size), but this potential benefit is offset by the need for a higher number of cores and higher communication costs. Traditional microkernels combine all components into a single server. However, a number of recent multikernels have argued for a split design [6, 24]. The design of PIKA allows us to evaluate the design trade-offs involved in each of these choices; we present the results in Subsection 5.4.

3.2 Speculative 3-Way Handshake

In monolithic kernels, the library function `accept()` is implemented using a shared accept queue within the kernel from which all applications can dequeue incoming connections; these queues are an example of shared state within the network stack that can become a scalability bottleneck [7, 18]. In PIKA, by contrast, each CM maintains its own accept queue, without cache-coherent shared memory to keep the queues coordinated. This fact presents a challenge in the implementation of `accept()`.

A naïve solution would have the CM enqueue all incoming connections; the application would then need to send a message to the CM on every invocation of `accept()`. In response, the CM would choose one of the enqueued connections and notify the owning TS to offer the connection to the application. This implementation, while straightforward, is not very performant. First, it adds the latency of up to three messages to every invocation of `accept()` ($App \rightarrow CM$, $CM \rightarrow TS$, $TS \rightarrow App$). An invocation of `select()` for a listen socket would similarly incur a round trip messaging cost to the CM. Each hit in a select loop would thus require five messages, while a miss would need two, leading to prohibitively high communication costs. More importantly, it can lead to very high message traffic as applications poll `select()`.

Our solution is to have the CM *speculatively* assign incoming connections to applications. Both `select()` and `accept()` become purely local operations: the application simply checks for new connection messages asynchronously forwarded by the CM. This scheme presents a challenge, as applications that either do not subsequently call `accept()` or happen to be busy lead to lost connections (above the TCP layer) or high latency. The CM must therefore keep an accurate list of ready applications so that most assignments are serviced quickly, and additionally provide a recovery mechanism in case the application does not claim the connection in a short time frame.

PIKA solves this challenge by employing a 3-way handshake between the application and the CM/TS. On the first invocation of `accept()` or periodically on invocations of `select()`, the application adds itself to the CMs list of acceptors by sending an `ACCEPT-CONNECTION` message. An incoming connection is then assigned to one of the acceptors and the TS is notified as such, causing the TS to send a `CONNECTION-AVAILABLE` message to the application. On the next invocation of `accept()` or `select()`, the application claims this new connection by sending a `CLAIM-CONNECTION` message to the TS, at which point the TS finishes connection establishment and confirms to the application that it successfully claimed the connection. If, however, the application does not respond to the `CONNECTION-AVAILABLE` message within a set period of time, the TS notifies the CM that the application timed out. In response, the CM removes the application from its list of acceptors and assigns the connection to another acceptor. If the original acceptor subsequently tries to claim the connection, it is notified that it timed out, at which point it needs to add itself again to the CM's list of acceptors by sending another `ACCEPT-CONNECTION` message. By using a small timeout, new connections can be efficiently reassigned from busy applications to ready applications.

The speculative 3-way handshake thus allows the CM to maintain an approximate list of acceptors that is lazily updated, greatly reducing the number of messages and improving latency relative to the naïve approach.

3.3 Load Balancing

PIKA CMs must distribute incoming connections among application processes, prioritizing those with the lowest service times. Ideally, CMs would know the immediate service times of all processes listening on a port and use these to select the best destination. This is unfortunately impossible due to the lack of cache-coherent shared memory. Instead, each CM maintains a private accept queue and attempts to model a globally shared queue via message passing. This presents several challenges: (i) maintaining accurate, shared state in a volatile environment, (ii) responding to load imbalance effectively without disrupting service time of other PIKA servers or applications, and (iii) doing so with minimal communication and synchronization overhead.

PIKA's load balancing scheme, described below, effectively balances incoming connections across all applications giving priority to local applications (i.e. processes on the private accept queue). It automatically handles the special case when a CM has no listening applications, acting as a connection distribution mechanism by allowing connections to be "stolen" by other CMs.

The general approach PIKA takes for balancing the load is to have CMs periodically (every 50 µs in the current implementation) update each other on the service times of their

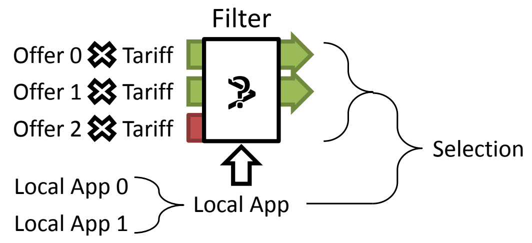


Figure 2: PIKA load balancing scheme. Local applications compete against *offers* from other CMs. Offers are only considered when significantly better than local applications. Competition occurs in rounds to scale gracefully.

applications. These updates are called *offers* and include the service time of the best application to receive connections (i.e., the fastest) along with its address (i.e., a handle) so it can be directly contacted by other CMs without any further coordination.

These service times are measured by keeping a window of the last several connections' service times. Service times are reported to CMs by the TSs when a connection is closed. Service time is defined as the time between *connection offered* and *connection closed*. This design captures end-to-end performance which includes a variety of load imbalances – such as an application co-scheduled with another process, applications servicing long-lived connections, applications on temperature-throttled cores, and so on. This metric may need to be changed to match the workload (e.g., for workloads with long-lived, low-utilization connections). Note, however, that load estimation is orthogonal to the remainder of the design.

Possessing the service times, CMs then compare their local applications with offers from other CMs to determine where to send incoming connections (Figure 2). Because service times are updated periodically, PIKA selects between applications probabilistically to prevent flooding. Each application is weighted by its inverse service time, favoring faster applications. This approach also lets the load balancer sample slow applications to see if they have recovered.

In order to encourage locality, which improves latency and reduces the number of messages, PIKA penalizes offers (i.e., non-local applications) by a multiplicative *tariff*. Offers are further filtered so that only those which are competitive with local applications are available for selection. This is done because there is no need for CMs to sample non-local applications (this will be done by the local CM), so there is never any benefit from choosing a non-local application with worse service time over a local one. Filtering allows PIKA to use low tariffs and still maintain locality.

PIKA selects between offers and local applications in a tournament of three rounds (Figure 2). This design addresses a scaling pathology that occurs with a single selection round. With a single round, the relative weight of local applications decreases with increasing numbers of offers. This means that as the number of CMs scales up, locality decreases. By using a tournament, the relative weight of local applications and offers is independent of the number of CMs.

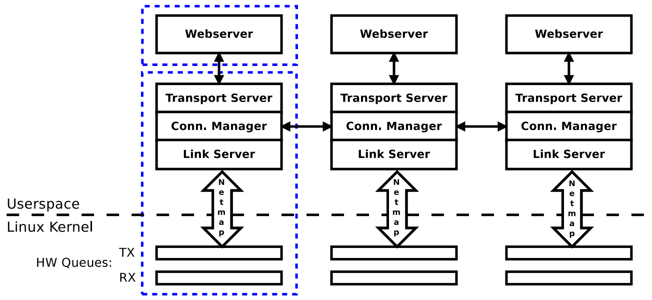


Figure 3: Diagram of PIKA implementation. All processes run independently using message passing as the only means for communication and synchronization. Dotted box indicates components running on a single core.

Finally, PIKA employs high- and low-watermarking of filtered offers to prevent thrashing (not shown in Figure 2). Once an offer has passed the filter, its performance relative to local applications must fall below a low-watermark value before it is filtered again.

4 IMPLEMENTATION

In order to facilitate PIKA's development, we implemented a multikernel emulation environment on top of the Linux kernel (Figure 3). This implementation allows PIKA to take advantage of high-performance Linux's NIC drivers, as well as many debugging and performance monitoring tools (such as `gdb`, `perf`, and Valgrind [4]). Note, however, that we simply use Linux to bootstrap the infrastructure and to map shared memory pages that are used solely by the user-space message passing library.

All application and server processes communicate and synchronize solely through message passing without reliance on cache-coherent shared memory and locking primitives above this messaging layer. Hardware DMA rings are mapped to distinct PIKA servers, so no communication occurs between cores in kernel-space (see Subsection 5.1). Process boundaries enforce communication via messages in user-space. Thus the system emulates a multikernel architecture. This design can easily be ported to hardware that does not support cache-coherent shared memory and other multikernels.

Alternatively, we could have simulated PIKA on a target architecture without cache-coherent shared memory. This approach would prevent any inadvertent use of cache coherence, but it has many problems. Modeling high-performance networking in a simulator is inherently difficult and error-prone. Because simulation runs orders of magnitude slower than native execution, faithfully modeling external network behavior (e.g., at the switch or client machines) under high load is impossible. Moreover, modeling the behavior of modern NICs is arcane and difficult to validate. Even disregarding these problems specific to PIKA, simulations of large multicore processors carry large margins of error and are difficult to validate. We therefore opted for a native implementation on commodity hardware and confirmed that no inadvertent use of cache-coherent shared memory takes place (Subsection 5.1).

Applications communicate with PIKA using the POSIX socket API. We use the GNU/Linux shared object interposition mechanism (through the `LD_PRELOAD` environment variable) to load a PIKA compatibility library that intercepts POSIX socket functions and translates them into messages to PIKA servers. Applications do not need to be modified or recompiled to use PIKA. This development infrastructure can be used to build a variety of multikernel services. We will release the source code publicly to encourage further development.

4.1 Driver and Network Stack Processing

The PIKA LS manages the NIC using netmap [19], which gives user-space programs device-independent access to the network interface. netmap maintains a copy of the device state in user-space, as well as a shadow copy in the kernel. The user-space application (i.e., the LS) modifies its view of the device state to enqueue packets for transmission or dequeue packets from a ring. The interface into netmap is through standard POSIX functions, which synchronize the driver state between the kernel and the application. The LS amortizes the costs of the system calls to interact with netmap by enqueuing and dequeuing batches of packets before each synchronization.

We use the version of netmap released in June 2012, which includes a modified ixgbe driver 3.3.8-k2 for the Intel 82599 10 Gigabit Ethernet Controller. We extend netmap to support additional features of our hardware, such as manipulating flow director tables and reading the MAC address.

Flow director is a feature of the Intel 82599EB 10 Gigabit Ethernet Controller that allows the software to control which hardware DMA ring an incoming packet is routed to based on attributes of the packet [2]. PIKA uses this feature to route packets to individual hardware DMA rings based on the low 12 bits of the source port number as suggested by Pesterev et al. [18]. Each hardware DMA ring is managed by a separate LS, ensuring that different connections are spread across the different servers while multiple packets for a given flow are always delivered to the same server.

The network and transport layer processing code (e.g., IP, TCP, etc.) employed by the PIKA TS is largely adapted from the lightweight TCP/IP (lwIP) stack [3].

4.2 Programming Model

PIKA implements a high-performance message passing library whose fundamental abstraction is a first-in first-out (FIFO) queue. Each queue is unidirectional; two queues are combined to provide a bidirectional communication medium.

PIKA servers are event-driven programs supported by a cooperative threading library and dispatcher. Each network stack component registers callbacks for unique message types and yields control to the dispatcher loop. Each callback is invoked in a cooperative thread with its own stack that can yield back to the dispatcher pending a response message from other PIKA servers or applications. The dispatcher provides its own message passing routines implemented on top of the base message passing library that handle all remote procedure call (RPC) traffic. The threading library implements microthreads [14] that avoid costly context switching. This model allows PIKA components to be easily combined into a single process avoiding additional context switches among components.

The message passing library runs on commodity hardware and therefore relies on cache-coherent shared memory. The queue abstraction is general enough, however, to be portable to hardware supporting message passing [15].

5 EVALUATION

This section evaluates these questions experimentally:

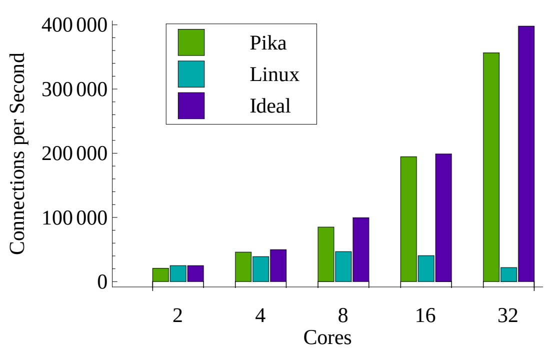
- Lacking cache-coherent shared memory, PIKA uses novel mechanisms to provide the illusion of a shared accept queue per port and to balance connections between applications. Does the overhead of maintaining this shared state affect the performance and scalability of PIKA with increasing core count?
- How well does PIKA's load balancing scheme maintain consistent service times? Can it adapt to changing workloads?
- To what extent do the various design trade-offs discussed in Section 3.1 impact the best configuration for PIKA, and how is the choice of best configuration affected by hardware parameters?

5.1 Experimental Setup

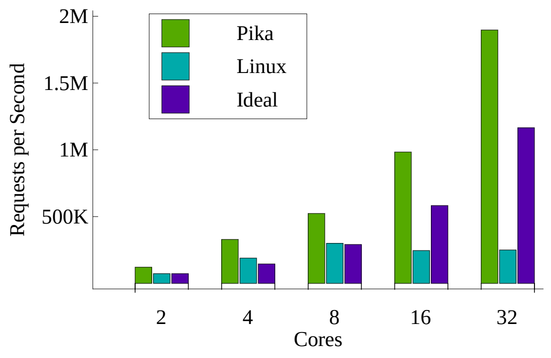
All of the results in this section are gathered on 4-socket, 40-cores PowerEdge 910 servers with four Intel Xeon E7-4850 processors and an Intel 82599EB 10 Gigabit Ethernet Controller. The machines run Ubuntu Server 11.10 x86-64 with the Linux kernel 3.0.0 and the "netmap-modified" ixgbe driver 3.3.8-k2. All machines are connected with an Arista 7124S 10 Gigabit Ethernet Switch.

In evaluating PIKA, we focus primarily on connection establishment, since it is the metric which depends the most on effective sharing of state for achieving good performance. Other metrics of network performance, such as bandwidth and small message throughput, are trivially parallelizable and are therefore not very interesting.

Our experimental setup is as follows unless stated otherwise. The application is a small multiprocess web server that serves pages from memory; this isolates the network stack and removes other potential performance bottlenecks. Since we wish to focus on connection establishment, all our experiments focus on short-lived connections (HTTP requests for a 4 B webpage without `keep-alive`). The client workload



(a) Connections



(b) Requests

Figure 4: Scaling of requests per second vs cores. Clients perform HTTP Gets on a 4 B webpage with and without `keep-alive`. Results show that PIKA scales ideally up to 32 cores. Linux is included for reference.

	DMA misses per req.	**Other misses per req.**	**Total kernel misses per req.**
Same socket	0.98	0.27	1.25
Across sockets	0.84	0.30	1.14

Table 1: Last-level cache misses in the kernel per HTTP request. Four PIKA servers and four web servers are run within a single socket or across different sockets. Cache misses do not increase across sockets; Linux is faithfully modeling a multikernel.

consists of 48 apachebench 2.3 [1] processes spread across four machines, each with 4 concurrent connections. For each result, clients are run for a sufficient time to capture steady state behavior. A single web server can saturate a TS, thus all of our configurations use an equal number of web servers and TSs. All processes are run on separate cores with communicating processes placed on the same socket where possible.

We provide comparisons to Linux kernel's network stack (Linux from now on) performance in many experiments to demonstrate that PIKA achieves good performance at low core counts. Due to the lack of other parallel network stacks that do not rely on cache-coherent shared memory, we evaluate PIKA in absolute terms against the network stack provided by Linux. To this end, we also include an "Ideal" configuration that ideally scales Linux's performance at two cores (the smallest system that can run PIKA) to larger systems. PIKA also compares well with results published for research network stacks employing cache-coherent shared memory [18], however we cannot compare directly as Affinity-Accept [18] does not run on recent versions of the Linux kernel.

Linux with netmap as a multikernel. By leveraging Linux on a cache-coherent shared memory multicore processor, one might worry that PIKA inadvertently benefits from cache coherence. To put this worry to rest, we measure the number of last-level cache (LLC) misses per HTTP request for a configuration of four combined PIKA servers and four web servers. Linux device interrupts for each hardware DMA ring are mapped to the corresponding PIKA server. The configuration is run with: all processes on the *same socket*; and server/application pairs striped *across sockets*. If any communication took place via cache-coherent shared memory to handle an HTTP request then one would see an increase in LLC misses when running across sockets.

Table 1 shows that LLC misses do not increase across sockets; thus, there is no hidden communication among cores.[2] The LLC misses that do occur are $\approx$ 1 miss per request in `ixgbe_netmap_rxsync` when accessing the hardware DMA ring, plus a small number of conflict or capacity misses elsewhere. Process boundaries ensure no inadvertent sharing occurs in user-space. The combination of Linux and netmap therefore behaves as a multikernel in our experimental setup.

In the remaining experiments, one core is dedicated to handling Linux device interrupts. Our driver uses polling, and device interrupts are only used to notify netmap of waiting packets in the hardware DMA rings. As a consequence, we do not consider this core in our evaluation.

Message Passing Performance. Table 2 measures interprocess communication in PIKA. The first experiment shows the limits of the cache coherence hardware using atomic memory instructions to receive a message and send a reply, taking on average 186 cycles per round-trip. For synchronous processes, this benchmark is the best possible result, because full-featured message passing must allow receivers to read a message and write a response. The remaining experiments evaluate PIKA's message passing and dispatching. With two synchronous processes, PIKA takes $\approx$ 600 cycles for a round-trip. The additional delay over baseline is due to allocation overhead and polling misses. However, with many communicating processes, these misses are hidden and PIKA matches the baseline performance at 150 cycles per round-trip. Dispatch adds modest overhead in all cases. Finally, creation, scheduling, and execution of a microthread that does not yield takes 26 cycles.

[2]Note that we disabled hardware prefetching to avoid false sharing.

	Benchmark	Socket	Latency	Cache misses
	Shmem Micro	On	186	2.00
		Off	964	2.00
Synchronous	Messaging	On	597	2.95
		Off	1898	3.13
	Dispatch	On	706	
		Off	2058	
Concurrent	Messaging	On	150	2.00
		Off	370	2.60
	Dispatch	On	378	
		Off	857	
	Microthread	–	26	

Table 2: Latency and cache misses for various programming model operations. Latency is shown for both on- and off-socket. Cache misses are an average for a single round-trip.

5.2 Scalability

This section evaluates the ability of PIKA to scale to meet demand with increasing core counts. To apply maximum stress to PIKA, we focus on shared state using a connection-per-second microbenchmark. By using short connections and small request sizes, we can isolate PIKA's scaling without any limitations from the application. As past research has shown, the majority of connections in real-world internet traffic is short-lived TCP connections [11]. We also present separate results for long lived connections, validating PIKA's performance at the other extreme.

The following results were gathered using the PIKA configuration that we empirically established as having the best performance (Subsection 5.3 and 5.4): the TS, LS and CM components combined in each server, and using the Tournament scheme with a tariff of 2 for load balancing.

Figure 4a reports the throughput obtained by PIKA for short-lived connections for various core counts. The same metric for Linux is included for reference. The data shows that PIKA achieves performance competitive with Linux for small core counts while significantly outperforming it for large core counts, matching the Ideal configuration's scaling. PIKA achieves near perfect scaling, chiefly because the use of individual accept queues by each CM removes the main source of contention in connection establishment.

The only possible non-architectural source of imperfect scaling is the messages exchanged by the CMs to maintain sharing semantics and to exchange service times. The number of these messages increases with increasing number of CMs. Table 3 shows the overhead for these updates. From this experiment, we can see that the number of updates per second increases as the number of CMs increases. However, the overhead for updating these local data structures is indeed negligible as only 0.34 % of time is spent on these updates in the worst case.

It is worth noting that Linux scales poorly due to lock contention on the shared accept queue and a lack of connection affinity to cores. Both of these problems have been addressed in Affinity-Accept [18], however the proposed solutions have not yet been incorporated into the Linux kernel.

Number of CMs	Updates / sec	Time updating	Overhead
2	21	27 μs	0.03%
4	63	65 μs	0.07%
8	147	131 μs	0.13%
16	315	342 μs	0.34%

Table 3: Time spent updating list of potential remote applications between Connection Managers over a 10-second test. In the worst case, updates incur 0.34% overhead.

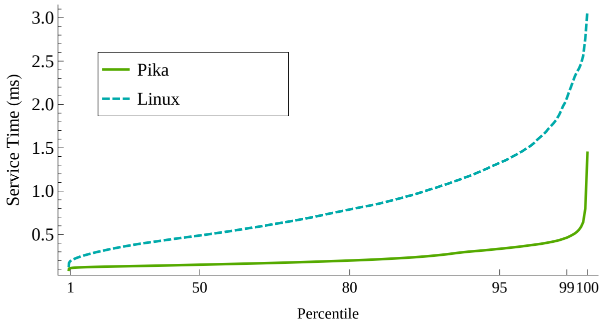
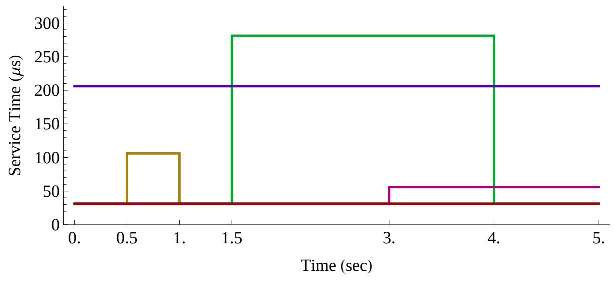


Figure 5: Service time percentiles for a connection/sec focused workload with load imbalance (see text). Results indicate that PIKA adapts to the load imbalance with minimal impact on service time (knee in curve occurs past 99^{th} percentile). Linux is included for reference. Note log scale on x-axis.

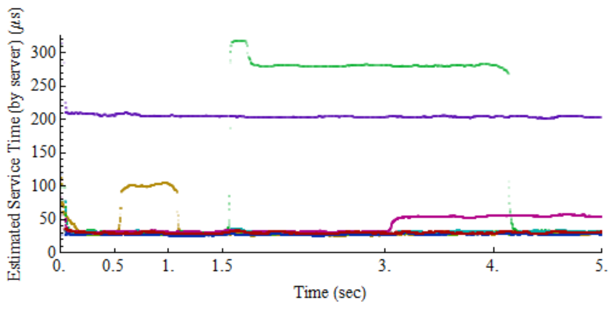
In terms of absolute performance, the throughput numbers reported in the Affinity-Accept paper on a high number of cores are roughly similar to the throughput numbers that PIKA achieves on a high number of cores. PIKA's throughput numbers are much better than previously published numbers for multikernels [6, 24] (i.e., 1.9 M reqs/s vs. 19 K reqs/s). However, any comparison between published results is complicated by the differences in experimental setup (e.g., processors, memory, NIC, etc.).

In addition to good scalability on connection establishment, Figure 4b demonstrates that PIKA achieves good performance and scalability on long-lived connections. As with short lived connections, PIKA achieves throughput that is competitive with Linux at small core counts while outperforming it at large core counts, even besting the Ideal configuration. While the handling of long-lived connections can be trivially parallelized, these results demonstrate that the various techniques PIKA uses to optimize connection establishment do not adversely affect its performance on long-lived connections.

The combination of these experiments demonstrate that PIKA performs well both in absolute terms as well as scalability for both short and long-lived connections, and thus it is our view that PIKA can achieve good performance across a wide range of workloads.



(a) Actual



(b) Measured

Figure 6: Webserver service times over the duration of the experiment. Three web servers experience transient delays of different magnitudes (yellow, green, magenta), while one web server remains consistently slow (purple). The four remaining web servers experience no artificial delay.

5.3 Load Balancing

This section evaluates PIKA behavior under an imbalanced workload and the trade-offs among different load balancing schemes. The key questions are:

- Does PIKA maintain consistent service times under an imbalanced workload?
- Does PIKA accurately and stably predict application service times?
- How well do different load balancing schemes respond to imbalance?
- What is the impact of different load balancing schemes on performance and scalability?

In order to evaluate different load balancing schemes in the presence of imbalance, we artificially introduce delays in the web server as shown in Figure 6a. The configuration is a 16-core system with 8 composite PIKA servers (TS, LS and CM combined) and 8 web servers. The system is driven by a client that maintains a constant request rate of 30K connections per second.

Given sufficient load, all applications can be kept busy by simply distributing incoming connections round-robin, ignoring application performance. So throughput, while important, does not indicate good load balancing. Worst-case or near-worst-case service time is an important metric for load balancing and can dominate overall response times in large scale parallel computing [5, 9]. We therefore focus on service time percentiles, particularly at the 95th percentile and above.

Figure 5 plots service time percentiles for PIKA and Linux. PIKA uses the Tournament scheme with a tariff of 2. PIKA has consistent service times with the knee in the curve occurring past the 99th percentile, indicating that PIKA can respond quickly and effectively to changes in service times of individual applications, avoiding slow applications and ensuring that overall service time percentiles do not suffer. Comparing the shape of the curve to Linux, PIKA does better at higher percentiles, showing that PIKA's tournament selection outperforms traditional first-come-first-serve allocation of a shared memory accept queue.

Accurate, stable, and responsive service time estimation is critical to the load balancer. Figure 6b shows PIKA's estimates of service time over the duration of the experiment. This is nearly identical to the actual service times (Figure 6a), demonstrating that PIKA has accurate knowledge of the service times of each application. In these experiments, we used a moving median with a window size of 128 and a sample interval of 500 µs. We found that a fairly large range of values worked well; for example, very similar data was generated using a moving average with a window size of 256 and sample interval of 1 ms.

Next we compare the following load balancing schemes: Tournament 2, Filtered 2, Unfiltered 2, Unfiltered 100, and Naïve, where the number indicates the tariff in all cases. Naïve always gives an incoming connection to a local application. Unfiltered X does not filter out offers from other CMs and performs a single selection round. Filtered X is similar to Unfiltered X, except all CMs whose offers do not outperform local applications with the tariff are ignored. Tournament X is the complete load balancing scheme described previously, including watermarking except where noted. The important measures are performance, as reflected in scalability; and quality of service, as reflected in service time percentiles.

Service Time. Figure 7 shows how well each scheme responds to the same imbalanced workload. Naïve's service time quickly degrades after the 80th percentile. Because Naïve never re-distributes connections, it always suffers the service time of its local application. Naïve has the steepest growth of any scheme, including Linux. Naïve represents a straightforward use of non-cache-coherent systems (e.g., running many independent Linux instances). Figure 7 shows that in the face of load imbalance, PIKA *greatly outperforms systems without coordinated load balancing.*

Unfiltered 100 performs poorly relative to the other schemes as well, with its service time rising sharply before the 99th percentile and diverging from the other schemes at the 80th percentile. This reflects the high tariff, which heavily penalizes distribution of incoming connections to other CMs. This prevents Unfiltered 100 from responding except

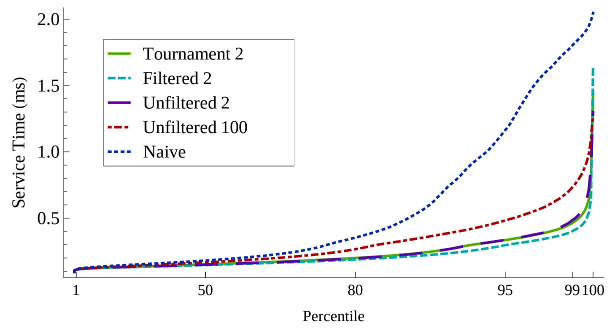


Figure 7: Service time percentiles for different load balancing schemes under an imbalanced workload. Tournament 2, Filtered 2, and Unfiltered 2 perform similarly and degrade little until the 99th percentile. Unfiltered 100 and Naïve perform poorly, diverging before 80th percentile.

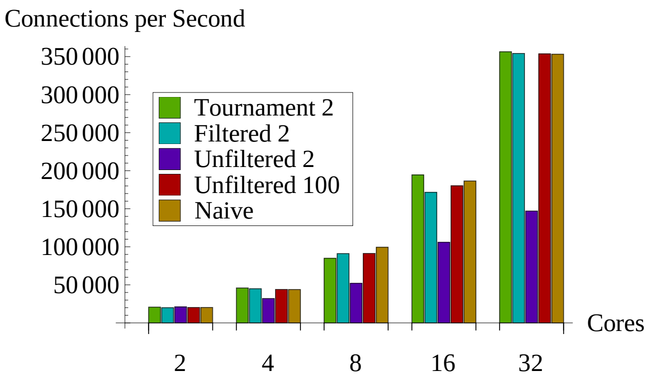


Figure 8: Scaling of connections per second with different load balancing schemes under a uniform workload. Naïve performs no load balancing – connections are always sent to the local web server. All load balancing schemes except Unfiltered 2 match naïve scaling.

in the most extreme cases of load imbalance, and it responds hardly at all to minor imbalances like those at 0.5 s and 3 s (Figure 6a).

The three remaining schemes – Tournament 2, Filtered 2, and Unfiltered 2 – perform nearly identically in this metric. All have low growth in service time before the 99th percentile and their service times are nearly identical.

Scalability. Figure 8 shows the scalability of the five load balancing schemes *under a uniform workload*. These results show the impact of load balancing on a balanced workload, where the proper response is to *not* balance.

As expected, Naïve performs well as the uniform workload does not require load balancing. Unfiltered 100 also does well, as the large tariff prevents connections from going to other CMs except in rare cases. Unfiltered 2, however, performs very poorly. With a low tariff, connections are often given to other CMs despite the uniform workload. This reduces locality and increases communication overheads, leading to a performance loss of over 2× at 32 cores. Filtering offers

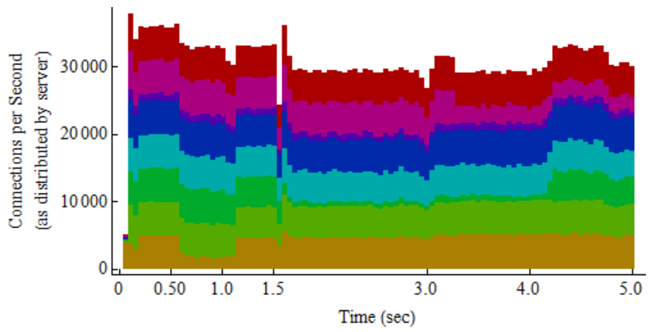


Figure 9: Throughput in connections/sec vs. time for an imbalanced workload. Throughput is partitioned by which web server received the connection. Ticks along the *x*-axis indicate changes in connection distribution as PIKA responds to imbalance.

that do not beat the tariff addresses this problem, as shown by Filtered 2 and Tournament 2, which match Naïve scaling and performance.

These results, combined with the previous section, demonstrate that Filtered 2 and Tournament 2 are the only schemes that achieve good scalability and service times under a variety of workloads.

Temporal Analysis. This section evaluates Tournament 2 over the test duration, showing how it responds to load and improves upon Filtered 2 in some cases.

Figure 9 shows the behavior of the complete system during the experiment. System throughput is plotted on the *y*-axis versus time. Throughput is divided by the number of connections distributed to each application, represented by the same colors as in Figure 6a. Ticks along the *x*-axis show where imbalance changes. Throughput is maintained at request rate throughout the experiment, with minor fluctuations when the load changes.

The response to load balance is clearly evident in the changing distribution of connections shortly after each tick. At 0.5 s, web server #1 (yellow) slows and receives sharply fewer connections. At 1 s web server #1 recovers and its throughput regains its previous rate. Similarly, web server #3 (green) experiences a drop and recovery at 1.5 s and 4 s, respectively.

In order to demonstrate the value of the Tournament 2 load balancing scheme over the other schemes we have plotted the number of connections that are assigned *locally* for web server #7 (magenta), the one that slows slightly at 3 s, for the workload in Figure 10. For Unfiltered 2 the tariff is inadequate to keep connections local, and it distributes connections to remote connection managers for the duration of the experiment yielding poor locality. Filtered 2 keeps connections local while the web server is performing on par with the others, however when the service time drops slightly at 3 s, it greatly prefers remote web servers much like Unfiltered 2. Tournament 2 performs the best since it keeps connections local until 3 s, after which it shows only a slight decrease in the

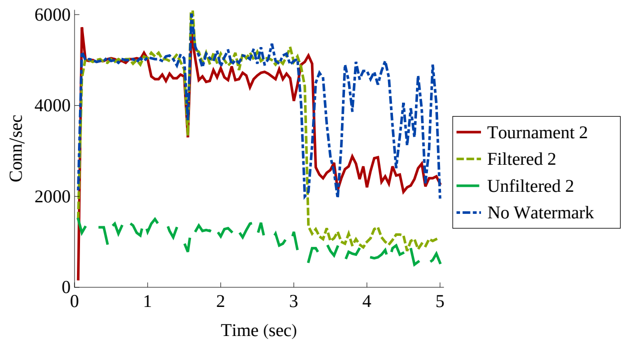


Figure 10: A comparison of different load balancing schemes for web server #7 (pink in previous figures) which has small, transient slowdown starting at 3 s. See text for discussion.

number of local assignments, reflecting the slight increase in service time. Due to multiple selection rounds, it still prefers to assign locally and produces a similar response with two CMs as with sixteen. Lastly, watermarking prevents thrashing of the filter in Tournament 2.

Discussion. PIKA's load balancer responds well to application demand without degrading scalability. With an imbalanced workload, naïve load balancing quickly degrades performance to unacceptable levels. Simple load balancing schemes are sufficient to address this, but filtering is necessary to avoid performance degradation under heavy, uniform load. The basic filtering scheme has undesirable scaling behavior, however, disfavoring locality as the number of CMs increases. This issue is addressed using a Tournament selector, which allows a single tariff value to scale up to a large number of CMs.

5.4 Configurations

In any multikernel architecture an important decision to make is how to split the system services and how many instances of each component should be run. Other multikernels have proposed splitting the network service for both performance and reliability reasons [6, 16, 24]. As discussed in Section 3.1, a split design affords better control over parallelism (with the attendant ease in managing shared state) and may also provide cache benefits. This comes at the price of higher communication costs and a larger core budget. We use PIKA's flexible design to evaluate these trade-offs.

We describe configurations in the following notation:

1. X_nY_n implies that component X is combined with (i.e., in the same process as) component Y, and there are n instances of this combined server.
2. X_nY_m ($m \leq n$) implies that there are n servers implementing the component X, and m of those also have component Y attached.
3. $X_n + Y_m$ implies that the system has n instances of component X and m instances of component Y (in separate processes).

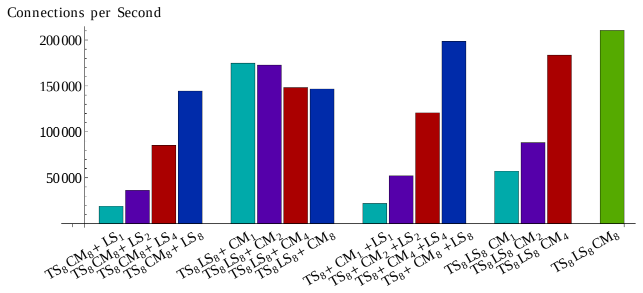


Figure 11: Comparison of different configurations with TS_8 held constant. Additional cores are used as LS, CM, or both. The combined configuration outperforms all others, while using fewer cores.

As an example, the split system architecture diagram (Figure 1a) depicts the configuration $TS_2 + LS_2 + CM_1$, which uses 5 cores, while the combined system architecture diagram (Figure 1b) depicts the configuration $TS_2LS_2CM_2$, which uses only 2 cores.

Combined is Best. To evaluate the impact of using a split design, we fix the number of TSs and applications at 8 cores each while varying the number of LSs and CMs. This experiment offers the most favorable evaluation for a split design, as additional cores used by the LS and CM are "free". Figure 11 shows the performance in connections per second from holding the number of TSs constant at 8 and adding "free" cores for other components as required by the configuration.

The far right bar shows the performance of the baseline configuration where all of the components are combined, thus using the least number of cores. Results are divided into four groups, all of which perform worse than the combined configuration:

- *Separate LSs.* Performance is very poor until LS_8. This is unsurprising: since the hardware has sufficient parallelism, lowering the number of LSs makes them the bottleneck. Separate LSs also suffer from additional message passing overhead for each TCP/IP packet.
- *Separate CMs.* Separating out the CM also degrades performance, although in this case not substantially from the baseline.
- *Separate LSs and CMs.* Given prior results it is expected that adding both LSs and CMs provides no net benefit, although $TS_8 + LS_8 + CM_8$ is close to the baseline (but using twice as many cores for the full system).
- *CMs in a subset of TSs.* The subset configuration depicted in the last group shows acute performance degradation at CM_1 and CM_2. This configuration reduces the overhead incurred from updates between connection managers; however, our results demonstrate that this gain is not enough to improve overall performance.

In conclusion, the baseline (combined) configuration outperforms any split configurations while using fewer cores. Since

Configuration	T_{CM}	T_{busy}	T_{msg}	T_{total}
$TS_1LS_1CM_1$	0.31 μs	3.13 μs	7.12 μs	307 μs
$TS_1LS_1+CM_1$	9.24 μs	3.67 μs	7.34 μs	311 μs

Table 4: Time spent establishing a connection with split or combined CM with a single client. Time spent in connection establishment increases significantly with a split CM.

Configuration	T_{CM}	T_{busy}	T_{msg}
$TS_8LS_8CM_8$	0.39 μs	3.26 μs	10.05 μs
$TS_8LS_8+CM_1$	17.78 μs	5.17 μs	8.61 μs
$TS_8LS_8+CM_2$	14.73 μs	4.96 μs	8.59 μs
$TS_8LS_8+CM_4$	13.67 μs	4.99 μs	10.05 μs
$TS_8LS_8CM_1$	6766.68 μs	5.91 μs	10.91 μs
$TS_8LS_8CM_2$	39.73 μs	5.92 μs	11.57 μs
$TS_8LS_8CM_4$	10.46 μs	5.18 μs	12.38 μs

Table 5: Time spent establishing a connection over all CM configurations using a full client workload. Time spent in the CM increases significantly with all split configurations.

messaging overhead is low and pipeline parallelism is ample, it is surprising that using more cores does not provide an increase in performance. We now take a more detailed look at where the time is being spent and cache behavior to determine the performance trade-offs.

Performance Breakdown. To better understand the performance trade-offs of running various components separately, we instrumented our code to measure the cost of different phases of handling an incoming connection. In particular, we measure the connection management cost (T_{CM}), defined as the time spent between the time a `SYN` packet is received by the LS to the time the packet is handed off to the TS to manage the connection. We also measure the time spent in the TS doing TCP/IP processing (T_{busy}) and in the message passing code (T_{msg}).

To estimate the messaging cost incurred by a separate CM, we compare the costs for $TS_1LS_1CM_1$ and $TS_1LS_1+CM_1$, where the application is a single web server and the client is an apachebench process with concurrency of 1 (Table 4) . The significant difference between the two is in T_{CM} (8.93 μs). This represents the additional communication cost incurred in $TS_1LS_1+CM_1$ and accounts for the slightly higher end-to-end latency (T_{total}) observed.

Table 5 presents these costs for all CM configurations, with the full experimental setup shown in Figure 11. Again, the most significant differences are in T_{CM}. The difference in T_{CM} between $TS_8LS_8+CM_1$ and $TS_8LS_8CM_8$ is higher than can be accounted for by the communication cost alone. The balance is the queueing delay experienced when a single CM is overwhelmed doing connection management for 8 TSs. The queuing delay decreases as the number of CMs is increased, though not enough to be an advantage while still incurring a communication overhead as well as using additional cores.

In the subset configuration, T_{CM} is several orders of magnitude higher for $TS_8LS_8CM_1$. In this case, the connection management time is made worse by the fact that the single CM additionally competes with heavily loaded TS/LS components, causing the queue length at the CM to grow unbounded. These delays explain the reduced throughput for this configuration observed in Figure 11. This time is decreased as the number of CMs is increased, however it remains much higher than in the combined configuration.

These results indicate that splitting the CM into a separate server or running it in a subset of the TSs does not provide any performance gains on our hardware due to queueing delays and additional communication cost.

Cache Size. One of the main motivations for splitting components is to reduce the cache footprint of PIKA servers so they fit in the private cache levels. To determine the cache impact of splitting the various components of PIKA, we simulate PIKA behavior for two configurations ($TS_1LS_1CM_1$ and $TS_1+LS_1+CM_1$) with a fixed workload of 10,000 requests for various cache sizes using Cachegrind [4]. Figure 12 presents the number of cache misses per request incurred for various cache sizes by these two configurations.

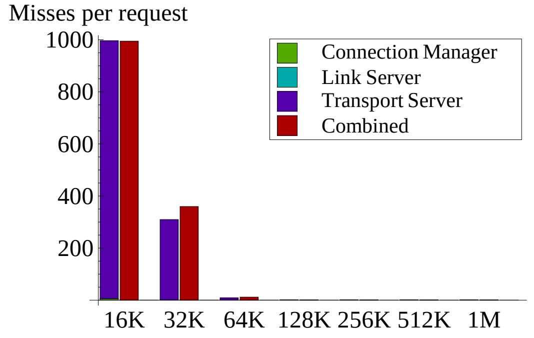
The results indicate that the total instruction footprint of all PIKA components fits in a cache size of 64 KB (Figure 12a). Thus, there is no advantage in splitting up the components for cache sizes over 64 KB. For smaller caches, the number of cache misses in the two cases are very similar; therefore any possible benefit due to components splitting, if at all present, is likely to be small.

For data caches (Figure 12b), the total cache misses incurred by the two configurations are very similar for caches bigger than 16 KB, and there are unlikely to be significant cache benefits in splitting the system. For cache sizes of 16 KB or smaller, the data indicates that there might be some advantageous cache effects to be obtained from splitting the components.

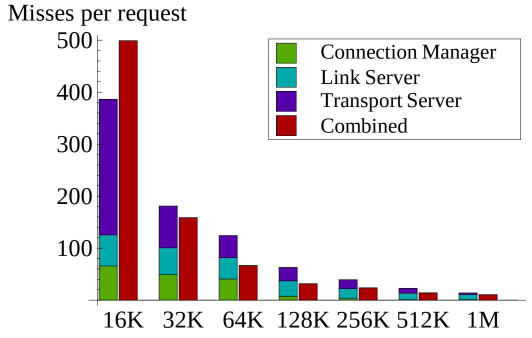
In summary, for systems with private caches larger than 64 KB, splitting the components is unlikely to result in better cache characteristics. The combined configuration ($TS_nLS_nCM_n$) is therefore better on commodity hardware, and likely on most multicore and many core processors in the foreseeable future. For smaller cache sizes or other system services with larger footprint, the choice is less clear and the answer depends on the relative importance of the various factors discussed above.

6 CONCLUSION

This paper presented PIKA, a network stack for multikernel operating systems. PIKA maintains a shared accept queue and performs load balancing through explicit message passing, using a speculative 3-way handshake protocol and a novel distributed load balancing scheme for connection acceptance.



(a) i-cache size



(b) d-cache size

Figure 12: Cache misses vs. private cache size. For private cache sizes larger than 64 KB it is preferable to combine components.

Our prototype achieves good performance, scalability, and load balance on 32 cores for both uniform and skewed workloads. Moreover, an exploration of possible multikernel network stack designs suggests that a combined network stack on each individual core delivers the best performance on current "big" cores, and that splitting the stack across several cores would make sense only on cores with 16 KB private caches. We hope that our techniques and results can inform the design of other system services for multikernels that must maintain shared state on future multicore processors.

Acknowledgments

This research was supported by Quanta.

References

[1] ab - Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] Intel 82599 10 GbE Controller Datasheet. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf`.

[3] lwIP - A Lightweight TCP/IP stack. `http://savannah.nongnu.org/projects/lwip/`.

[4] Valgrind. `http://www.valgrind.org`.

[5] Luiz André Barroso. Warehouse-Scale Computing: Entering the Teenage Decade. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[6] Andrew Baumann, Paul Barham, Pierre Évariste Dagand, Timothy L. Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[7] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[8] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. Dynamic Load Balancing on Web-Server Systems. *IEEE Internet Computing*, 3(3), 1999.

[9] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Commun. ACM*, 56(2), 2013.

[10] Daniel Grosu. *Load Balancing in Distributed Systems: A Game Theoretic Approach*. PhD thesis, University of Texas San Antonio, 2003.

[11] Liang Guo and Ibrahim Matta. The War between Mice and Elephants. In *Proceedings of the 9th International Conference on Network Protocols (ICNP)*, 2001.

[12] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[13] Nikhil Handigol, Srinivasan SeetharamanâĂă, Mario Flajslik, Nick McKeown, and Ramesh Johari. Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow. `http://conferences.sigcomm.org/sigcomm/2009/demos/sigcomm-pd-2009-final26.pdf`, 2009.

[14] Joe Hoffert and Kenneth Goldman. Microthread - An Object Behavioral Pattern for Managing Object Execution. In *Proceedings of the 5th Conference on Pattern Languages of Programs (PLoP)*, 1998.

[15] Jason Howard, Saurabh Dighe, Sriram R. Vangal, Gregory Ruhl, Nitin Borkar, Shailendra Jain, Vasantha Erraguntla, Michael Konow, Michael Riepen, Matthias

Gries, Guido Droege, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek K. De, and Rob F. Van der Wijngaart. A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling. *J. Solid-State Circuits*, 46(1), 2011.

[16] Tomas Hruby, Dirk Vogt, Herbert Bos, and Andrew S. Tanenbaum. Keep Net Working - On a Dependable and Fast Networking Stack. In *Proceedings of the Annual International Conference on Dependable Systems and Networks (DSN)*, 2012.

[17] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why On-Chip Cache Coherence Is Here to Stay. *Commun. ACM*, 55(7), 2012.

[18] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*, 2012.

[19] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the Annual Technical Conference (USENIX ATC)*, 2012.

[20] Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. IsoStack – Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of the Annual Technical Conference (USENIX ATC)*, 2010.

[21] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load Distributing for Locally Distributed Systems. *IEEE Computer*, 25(12), 1992.

[22] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[23] Livio Soares and Michael Stumm. Exception-Less System Calls for Event-Driven Servers. In *Proceedings of the Annual Technical Conference (USENIX ATC)*, 2011.

[24] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *Proceedings of the 1st Symposium on Cloud Computing (SoCC)*, 2010.