Corey: an operating system for many cores

Silas Boyd-Wickizer[*] Haibo Chen[†] Rong Chen[†] Yandong Mao[†]
Frans Kaashoek[*] Robert Morris[*] Aleksey Pesterev[*] Lex Stein[‡] Ming Wu[‡]
Yuehua Dai[°] Yang Zhang[*] Zheng Zhang[‡]

[*]MIT [†]Fudan University [‡]Microsoft Research Asia [°]Xi'an Jiaotong University

ABSTRACT

Multiprocessor operating system kernels typically provide complex abstractions implemented with shared data structures protected by locks. On multicore systems this design may cause the kernel to be a bottleneck due to the costs of contention for shared data and locks, and the costs of inter-core TLB invalidation.

Corey is a new operating system based on the principle that *applications should control all sharing*: all kernel data structures should be local to a processor core unless directed otherwise by the application. Corey is an exokernel-like operating system with new low-level abstractions (shares, address trees, and kernel cores) that allow library operating systems and applications to control all inter-core sharing. Corey also takes advantage of the likely abundance of cores by allowing library operating systems to dedicate cores to handling specific functions and data.

A Corey prototype runs on 16-core AMD and Intel machines. Measurements of MapReduce and Web server applications demonstrate that Corey can scale better and achieve better performance than Linux (9% for a MapReduce application, and a factor of 50% for a kernel-intensive synthetic Web benchmark). Microbenchmarks and performance counters confirm that these improvements are due to avoiding operations that are expensive on multicore machines.

1 INTRODUCTION

Multicore architectures, which have several processing cores on a die, have been adopted by most chip manufacturers. Chips with two and four cores are common, and trends suggest that chips with tens to hundreds of cores will appear within five years [2]. How should an operating system be designed to scale to a large number of cores?

One answer is to keep scaling existing operating systems. Many existing operating systems share complex kernel data structures among all processors, and have evolved to handle more and more processors by means of fine-grained locking and careful design of data structures to reduce contention. This approach has been successful, but is likely to become progressively harder with more cores due to contention. For example, Gough *et al.* show that contention for Linux's scheduling queues can contribute significantly to total application run time on two cores [12]. Veal and Foong show that directory lookups can cause a Linux Web benchmark to saturate the interconnect between cores [22]. Section 7 shows that Linux on 16 cores can have poor network performance when processing many short connections due to contention for shared network structures in the kernel. Section 7 also shows that a shared page table can limit the scalability of MapReduce applications, because of the cost of TLB shootdowns and soft page faults.[1]

This paper explores the design space from the other direction, with a single new principle: applications should control all sharing. This principle has two implications. First, the kernel should use only local data structures on each core, to avoid introducing unwanted sharing. Second, the kernel should present interfaces that let applications make the trade-offs between the benefits and costs of sharing. The intended result is that applications can tailor their use of processors and shared memory to achieve maximum performance.

The Corey kernel, described in this paper, embodies the principle of application control. It exports low-level interfaces to hardware resources such as physical CPUs, memory, and devices, much as in the exokernel [10]. A major design challenge lies in the choice of kernel abstractions: they must be implementable without inter-core sharing by default, but must be sharable among the kernels on different cores when requested by applications. *Shares* allow applications uniform and precise control over which kernel resources are shared between cores. *Address trees* allow applications to control which page-table entries are private per core and which are shared; manipulating the former requires less contention and fewer inter-core TLB invalidations, while the latter incur fewer total soft page faults when lazily constructed. *Kernel cores* allow an application to dedicate physical cores to run the kernel, reducing contention and enabling system calls via fast shared-memory IPC rather than slow traps. Corey allocates physical cores to applications rather than presenting a time-shared virtual pro-

[1]A soft page fault occurs when a core references a page that is present in memory and mapped in the software page table of the address space, but not yet loaded into the hardware page table.

cessor abstraction.

When sharing is required, it is often provided as part of a standard set of user-level "library operating systems." Corey's default library operating system provides a new scalable shared buffer cache design. It provides a new *cfork* function that allows an application to use a new core while precisely controlling the sharing of hardware resources through shares and address trees. The default library operating system provides a separate TCP/IP stack per core to reduce contention, but efficiently shares hardware network interfaces.

A comparison of two applications (MapReduce and a Web server) on Corey and Linux illustrates the benefits of application-controlled sharing. On 16 cores, MapReduce is 9% faster on Corey than on Linux. Performance counters and microbenchmarks show that the difference is due to a reduction in TLB shootdowns. The Web server uses the Corey kernel core abstraction to dedicate cores to networking functions. Measurements show that this dedication leads to 50% higher throughput than Apache on Linux. TCP short-connection microbenchmarks show that kernel cores and shares allow Corey to scale perfectly with the number of cores, while Linux's performance degrades. Finally the locality-aware Corey buffer cache scales perfectly for small reads, while Linux's performance plateaus at one core.

The contributions of this paper are (1) new operating system abstractions that allow applications to control sharing (e.g., shares, address trees, kernel cores); (2) the default library operating system design; (3) library extensions for Web server and MapReduce applications that exploit the ability to control sharing; and (4) an evaluation on AMD and Intel 16-core machines that demonstrates that application-controlled sharing can improve performance.

This paper should be viewed as making a case for application-controlled sharing, rather than demonstrating any absolute conclusions. Corey is an incomplete prototype, and thus it may not be fair to compare it to full-featured operating systems. In addition, the architecture of future multicore processors is a matter of speculation.

The rest of the paper is organized as follows. Section 2 identifies the operating system challenges posed by multicore processors. Section 3 presents the Corey kernel and Section 4 the default library operating system. Section 5 summarizes the extensions to the default library operating system to implement MapReduce and Web server applications efficiently. Section 6 summarizes the implementation of Corey for the AMD and Intel processors. Section 7 presents performance results of Corey running on 16-core AMD and Intel machines. Section 8 reflects on our results so far and directions of future work. Section 9 relates Corey to previous work. Section 10 summarizes our conclusions.

2 MULTICORE CHALLENGES

The main goal of Corey is to allow applications to scale well with the number of cores. This section details some hardware obstacles to achieving that goal.

Future multicore chips are likely to have large total amounts of on-chip cache, but split up among the many cores. Performance may be greatly affected by how well software exploits the caches and the interconnect between cores. For example, using data in a different core's cache is likely to be faster than fetching data from memory, and using data from a nearby core's cache may be faster than using data from a core that is far away in the interconnect.

These properties can already be observed in current multicore machines. Figure 1 summarizes the cache organization of 16-core machines from AMD and Intel. The AMD machine has four quad-core Opteron processors connected by a square interconnect. Each core has a private L1 and L2 cache. Four cores on the same chip share an L3 cache. Cores in one chip are connected by an internal crossbar switch and can access each others' L2 caches efficiently. Memory is partitioned in four banks, each connected to one of the four chips. Each core has a clock frequency of 1.9 GHz.

The Intel machine has four quad-core Xeon processors, each consisting of two dual-core dies. Each core has a private L1 cache. Each pair of cores on the same die shares an L2 cache. The four chips are connected via a shared front-side bus to the memory system. Each core has a clock frequency of 1.6 GHz.

Figure 1 also shows the cost of loading a cache line from a local core, a nearby core, and a distant core. We measured these numbers using many of the techniques documented by Yotov et al. [25]. The techniques avoid interference from the operating system and hardware features such as cache line prefetching.

Reading from the local L3 cache on an AMD chip is faster than reading from the cache of a different core on the same chip. Inter-chip reads are slower, particularly when they go through two interconnect hops. For the Intel machine, communication between cores on the same die is cheap due to the shared L2 cache, but communication between cores on different dies within the same processor is 4 times slower.

Figure 1 also shows costs for spin locks. The "Lock uncontended" line reports the measured cost of acquiring and releasing a spin lock that is in another core's L1 cache. For an AMD machine, two cores at 2-hops distance have a 27% performance penalty over 1 hop. For the Intel machine, two cores on different chips are more than 20% slower than those from the same chip.

The "Lock contended" line reports the average measured cost for 16 cores to acquire and release a contended

		AMD	Intel
Organization			
Cache line size		64 Bytes	64 Bytes
L1	cache size	64 Kbyte	32 Kbyte
	associativity	2-way	8-way
L2	cache size	512 Kbyte	2 Mbyte
	associativity	16-way	8-way
L3	cache size	2 Mbyte	N/A
	associativity	32-way	N/A
Memory References (cycles)			
L1 access		3	3
L2 access		14	14
L3 access		75	N/A
RAM		258	111
L1 same die		N/A	67
Local cache		127	63
Remote cache		205/285	110
Remote RAM		280/336	N/A
OS operations (cycles)			
Lock	uncontended	214/269	204/145/251
	contended	4799	1063
TLB	2 cores	4247	7224
	16 cores	21394	21270

Figure 1: Properties of AMD and Intel machines. Cache access latencies measured in cycles. Local cache latency is measured between two cores on the same processor (between different dies on the Intel). Remote cache access values for the AMD correspond to cores one hop and two hops apart on the system bus. Intel uncontended lock numbers are for same die, same processor, and remote processor lock accesses. N/A stands for not applicable.

spin lock (with no work in the critical region). A contended lock is 20 times as expensive on an AMD machine than an uncontended lock and 4 times as expensive for an Intel machine.

The "TLB" measurements give the average times required to send an interrupt to all other cores to ask them to flush their TLBs after a change to a shared page table. The TLB shootdown process for 16 cores versus 2 cores is 5 times as expensive on an AMD machine and 3 times as expensive on an Intel machine.

On the AMD and Intel 16-core systems, the speed difference between a fast memory access and a slow memory access is a factor of 100, and that factor is likely to grow with the number of cores. To get good performance, kernels must mainly access data in the local core's cache. A contended (or falsely shared) cache line will decrease performance because many accesses will be to remote caches. Widely-shared and contended locks will also decrease performance, even if the critical sections they protect are only a few instructions. This performance reduction will grow as the number of cores increases, since the cost of accessing far-away caches will increase. As mentioned in Section 1, several performance studies of Linux have shown that remote accesses, kernel locks, and TLB operations as part of kernel functions can become a bottleneck for application performance.

3 COREY KERNEL

The design of Corey is driven by the principle that applications should control sharing. By default, the Corey design ensures that cores use only local resources and share no state (including kernel state) between cores. Applications create sharing as needed using the Corey kernel interface.

To give applications full control over sharing, Corey is an exokernel: its role is limited to managing and protecting hardware resources. Applications use user-level "library operating systems" to implement more traditional operating system interfaces and can replace or reimplement subsections of the library operating systems as needed.

Figure 2 provides an overview of the Corey kernel interface. Corey exposes hardware resources as five types of objects: shares, segments, address trees, pcores, and devices. The following sections describe the design of the five Corey kernel objects and how they allow library operating systems to control sharing.

3.1 Object metadata

The kernel maintains metadata describing each object. To reduce the cost of allocating memory for object metadata, each core keeps a local free page list. The system call interface allows the caller to indicate which core's free list a new object's memory should be taken from. Kernels on all cores can address all object metadata since each kernel maps all of physical memory into its address space.

Corey uses spin locks and read-write locks to synchronize access to object metadata. As described in the following section, library operating systems specify all instances when kernel state is shared between cores, and therefore control when cores contend for locks and kernel data.

3.2 Object naming

A library operating system allocates a Corey object by calling the corresponding `alloc` system call, which returns a unique 64-bit object ID. In order for a kernel to use an object, it must know the object ID (usually from a system call argument), and it must map the ID to the address of the object's metadata. Since Corey aims to provide library operating systems full control over sharing, a global table mapping IDs to addresses would be an unacceptable bottleneck. Instead every core uses a set of lookup tables, known as *shares*, which map object IDs to metadata addresses. Only objects contained in a core's shares can be directly manipulated by that core. Library operating systems specify which shares are available on

System call	Description
obj_get_name	return the name of an object
share_alloc	allocate a share object
share_addobj	add a reference to a shared object to the specified share
share_delobj	decrement reference count of a shared object in the specified share
segment_alloc	allocate physical memory and return a segment object for it
segment_copy	copy, copy-on-write, or copy-on-read a segment
segment_get_nbytes	get the size of a segment
segment_set_nbytes	set the size of a segment
at_alloc	allocate an address tree object
at_get	return the address mappings for a given address tree
at_set	set the address mappings for a given address tree
at_set_slot	insert one address mapping into the specified address tree
pcore_alloc	allocate a physical core object
pcore_current	return a reference to the object for current pcore
pcore_run	runs this core with the specified user context
pcore_add_device	specify device list to a kernel core
pcore_set_interval	set the time slice period
pcore_halt	halt the pcore
device_list	return the list of devices
device_alloc	allocate the specified device and return a device object
device_stat	return information about the device
device_conf	configure a device
device_buf	feed a segment to the device object
self_drop	pcore drops its reference to a share
locality_get	get hardware locality information

Figure 2: Corey system calls.

which cores by passing a core's share set to `pcore_run` (see below).

When allocating an object, a library operating system selects a share to hold the object ID. The library operating system uses $\langle$share ID, object ID$\rangle$ pairs to specify objects to system calls. The kernel counts references to objects. Library operating systems can link an object to a share with `share_addobj`, incrementing its reference count, and remove an object from a share with `share_delobj`, decrementing its reference count. When an object reference count is zero, the object is no longer accessible by any core, and the kernel reclaims it. The kernel also counts references to shares, but the reference count of a share is equal to the number of cores that are using it.

By convention, library operating systems maintain per-core private shares and one or more global shares. Per-core shares hold private objects that are never shared, such as the stack segments for threads pinned to that core. Global shares facilitate inter-core sharing. For example, the library operating system inter-core dynamic memory allocator uses a global share for its memory segments and address trees.

3.3 Memory management

The kernel represents physical memory using the *segment* abstraction. Library operating systems use `segment_alloc` to allocate segments and `segment_copy` to copy a segment or mark the segment as copy-on-reference or copy-on-write. By default, only the core that allocated the segment can reference it; a library operating system can arrange to share a segment between cores by adding it to a share, as described above.

A library operating system uses *address trees* to define its address space. Each running core has an associated root address tree object containing a table of address mappings. Corey supports two types of address mappings. One is from virtual addresses to segment objects. The second is from virtual addresses to internal address trees. Internal address trees can only map segments.

This design gives library operating systems control over which part of a page table is private to a core (virtual address to segment mappings) and which parts of a page table is shared among cores (shared internal address trees). As we demonstrate in Section 7, the conventional approach of completely shared page tables or completely separate page tables is too coarse for multicore environments and can impede scalability. Address trees provide library operating systems with fine-grained control over what is shared.

3.4 Execution

Corey represents physical cores with *pcore* objects. Once allocated, a library operating system can start execution on a physical core by invoking `pcore_run` and specifying a pcore object, instruction and stack pointer, a set of shares, and an address tree. A pcore executes until `pcore_halt` is called by the executing pcore or by another pcore. This interface allows Corey to space-multiplex applications over cores, dedicating a set of cores to a given application for a long period of time, and letting each application manage its own cores.

This design makes sense where there are more cores than a traditional operating system has processes, since it eliminates the cost of time-multiplexing applications over a set of cores (e.g., context switches). Additionally, it eliminates the kernel state required to manage scheduling, which has been a source of contention for traditional operating systems on multicore systems [12]. Library operating systems can implement their own finer-grained cooperative scheduling policies (e.g., the

Phoenix MapReduce [16] implementation has its own dynamically load-balancing scheduler) and, if necessary, use `pcore_set_interval` to implement preemptive scheduling policies. Library operating systems wishing to time-multiplex physical cores between applications executing in different address spaces can still do so (and do so efficiently) by using `pcore_run`.

3.5 Devices

The Corey kernel exports devices using device objects. A library operating system can discover which devices are present on the machine using `device_list` and open a device by calling `device_alloc` and specifying a unique device ID. Library operating systems have exclusive control over devices they open, and if necessary they can implement policies to multiplex the devices between several cores.

The interface between devices and library operating systems is similar to Xen's asynchronous I/O rings [4] and HiStar's network device interface [26]. A library operating system communicates with a device it has allocated through a segment shared with the device. The library operating system specifies the segment with `device_buf`. Each device segment contains a header (and possibly some data) that the Corey kernel device drivers interpret in a device-specific manner. For example, to transmit an Ethernet packet, a library operating system sets the transmit bit and packet length of the device segment header and copies the contents of the packet into the device segment. When the packet has been transmitted, the kernel device driver sets the transmit complete bit in the device segment header, which the library operating system can read to determine if it is safe to reclaim the device segment.

Currently Corey supports a network device object, a virtual network device object, a console device object, and a disk device object. All Corey Web server experiments discussed in this paper were configured with network device objects (and not virtual network device objects). The Corey library operating systems include two experimental persistent file systems that use the disk device object, but no results presented in this paper use either file system.

3.6 Kernel execution

Corey provides a "kernel core" abstraction to allow a library operating system to dedicate one or more physical cores to executing the kernel. A kernel core can manage hardware devices and execute system calls sent from other cores via shared-memory IPC. Using a dedicated core reduces contention costs for kernel objects (such as devices) that are accessed exclusively by a kernel core and eliminates user/kernel context switch time for system calls.

A library operating system configures a kernel core by allocating a pcore object, specifying a list of devices with `pcore_add_device`, and invoking `pcore_run` with the kernel core option. A kernel core continuously polls the specified devices by invoking a device specific function. A kernel core polls both real devices and special "syscall" pseudo-devices.

A *syscall device* allows a library operating system to invoke system calls on a kernel core. The library operating system communicates with the *syscall device* via a ring buffer in a shared memory segment.

A common use of kernel cores is to have a single core manage a hardware device such as a network card. A kernel core running on that core handles low-level device interaction, polls for received packets and transmits completion indications. The kernel core also has a syscall device; library operating systems on other cores send and receive packets with `device_buf` system calls sent via the syscall device. The benefit of this arrangement is that it eliminates contention for the device's low-level data structures and the locks that protect them. Library operating systems can execute concurrently with the system calls they send to the kernel core by doing other work while periodically monitoring the syscall device's ring buffer.

4 STANDARD LIBOS

This section describes the abstractions that are part of the standard Corey library operating system [10]. Because the library operating system runs in user space, it requires no special privileges to modify it, and every aspect can be changed. Section 5 describes two extensions that improve multiprocessor performance for specific applications.

Basic C library functions are provided by uClibc[2] and efficient inter-core dynamic memory allocation is provided by Streamflow [19]. The core library operating system services that provide execution forking, efficient file sharing, and network access are detailed in the following sections.

4.1 `cfork`

`cfork` is similar to Unix `fork` and is the main abstraction used by library operating systems to extend execution to a new core. `cfork` takes a physical core ID, allocates a new pcore object and runs it. By default, `cfork` shares little state between the parent and child pcore. The caller marks most segments from its root address tree as copy-on-write and maps them into the root address tree of the new pcore. Callers can share a segment or an interior address tree with the new processor by marking the mapping as shared. Applications implement fine-grained sharing using shared segment map-

[2] http://www.uclibc.org/

pings and more coarse-grained sharing using shared interior address tree mappings. `cfork` callers can share kernel objects with the new pcore by passing a set of shares for the new pcore.

4.2 Buffer cache

An inter-core shared buffer cache is critical to system performance and often necessary for correctness when multiple cores are accessing shared files. Since the buffer cache is shared between cores the data structures maintaining the buffer cache can have high contention. Furthermore, under write-heavy workloads it is possible that cached disk blocks will also be contended.

At a high level the Corey buffer cache resembles a traditional Unix buffer cache. Cached disk blocks are stored in a tree. On a disk read or write request, the tree is first searched for the requested block before issuing the disk request. The Corey block tree is lock-free and relies on hardware support for atomic compare-and-swap operations. This has the benefit that common case block tree lookup operations do not modify any state, so simultaneous read operations can execute without any cache contention caused by lock acquisition. In contrast, the block tree structure in Linux is protected by a read-write lock, so a lookup must acquire the lock in read mode. The Linux read-write lock implementation uses a single integer to represent lock state, so simultaneous read lock operations will contend on the lock state.

Writes to the buffer cache modify shared state, making it difficult for concurrent writers to avoid contention. If writes are not handled efficiently, write workloads will scale very poorly. In traditional systems, a write request is processed by copying application memory into blocks stored in a block tree. This can be a costly operation if the blocks being updated are not already stored in the updating core's cache. Corey tries to minimize contention on shared data and cache misses using per-core lock-free block allocators and by copying application memory into blocks likely to be held in local hardware caches.

On writes, blocks are allocated from a per-core lock-free stack. Each block has a generation number that is incremented each time it is reused. For a full block write, the written data is copied directly into a newly allocated block, and the block is inserted into the block tree. For a partial write, the current block's generation number and address are recorded, and the non-overwritten data is copied from the current block to create the new version. Before the block in the block tree is replaced, its generation number and address are compared to the recorded values. If they are equal, the block is replaced; otherwise the partial write is retried. This scheme does not require any locks and is optimized for full block writes. However, it is possible that partial writes must be retried multiple times.

The block tree frees old blocks by pushing them onto the per-core stack they were allocated from. Ideally, blocks at the front of the stack will be in the cache of the core that owns them, so writing to them will not cause cache misses. If blocks were recently read, they may be held in another core's hardware cache. However, cores only modify blocks they own, so block data will only be fetched from remote caches if the owner of the block does not have its contents cached. This property helps reduce traffic on the system bus.

The buffer cache must ensure that the blocks are not freed or reused during reads. Corey uses a per-file read-write lock to accomplish this. Before an application reads cached blocks, the library operating system pins the entire contents of the file by acquiring the lock in read mode. When a library operating system removes file contents from the buffer cache it acquires the lock in write mode.

One problem with a naïve read-write lock implementation is that it requires modifying a shared variable to record the lock state. Contention on this one cache line can degrade performance. The Corey read-write lock implementation avoids this problem by using a variable per core for readers and one variable for all writers. Readers avoid cache contention by setting the read variable associated with the current core. The locking operation required for write mode is more expensive; the writer must set the write variable and scan all read variables ensure there are no current readers. However, library operating systems only acquire the lock in write mode when removing content from the buffer cache, which is an infrequent operation.

4.3 Network

Following the exokernel philosophy, Corey only mediates access to the network hardware; applications are responsible for providing their own protocol stack. The default library operating system uses the lwIP[3] embedded networking library. Each instance of lwIP uses a different Corey network device. If the number of network stacks exceeds the number of physical network cards, Corey virtualizes the cards. Virtualizing the network card in the kernel requires some implicit shared state. The TX and RX DMA ring buffers of the network card are shared between network stacks, as well two spin locks serializing access to the ring buffers. The hardware virtualization capabilities of newer network cards (e.g. a configurable number of receive and transmit buffers) may allow this state to be eliminated.

Library operating systems can choose to run several networks stacks (possibly one for each core) or a single shared network stack. All configurations we have experimented with run a separate network stack for each

[3] `http://savannah.nongnu.org/projects/lwip/`

core that requires network access. This architecture provides excellent scalability but requires multiple IP addresses per server and must balance requests using an external mechanism. A potential solution is to extend the Corey virtual network driver to use ARP negotiation to balance traffic between virtual network devices and network stacks (similar to the Linux Ethernet bonding driver).

5 APPLICATIONS

It is unclear how big multicore systems will be used, but we think that the general areas of parallel computation and network servers are likely application areas for such systems. To evaluate Corey in these areas, we chose a MapReduce system as a parallel computation, and a Web server with synthetic dynamic content as a network server. Both applications are tailored to data-parallel applications but have sharing requirements and thus are not trivially parallelized. This section describes how we implemented the two applications on Corey and how we extended Corey's standard library operating system to take advantage of domain knowledge to achieve good application performance.

5.1 MapReduce applications

MapReduce is a framework for data-parallel execution of simple programmer-supplied functions [9]. The programmer need not be an expert in parallel programming to achieve good parallel performance. Data-parallel applications fit well with the architecture of multicore machines, because each core has its own private cache and can efficiently process data in that cache. If the runtime does a good job of putting data in caches close to the cores that manipulate that data, performance should increase with the number of cores. The difficult part is the global communication between the Map and Reduce phases.

We ported the Phoenix MapReduce implementation [16] to Linux to learn about multicore programming. Phoenix is a publicly-available MapReduce implementation optimized for shared-memory multiprocessors using the SPARC multicore chips, running Solaris.

We found that by careful programming we could improve Phoenix's performance substantially; for example, our optimizations improved the performance of our application by a factor of four on 16 cores. These optimizations include: lock-free task scheduling, a hashtable-based bucket implementation, linked-list based buffer resizing, local reduce, and a good memory allocator [19]. All these optimizations boil down to reducing global operations and using local caches effectively; that work helped us shape the ideas in this paper.

We also have ported the original and optimized version to Corey. On Corey Phoenix can exploit address trees.

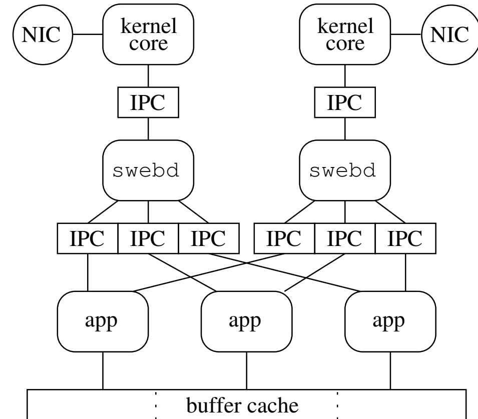


Figure 3: A Corey Web server configuration with two kernel cores, two `swebd` cores and three application cores. Rectangles represent segments, rounded rectangles represents pcores, and circles represent devices.

MapReduce applications can tax the virtual memory system by allocating gigabytes of physical memory to hold intermediate results, taking hundreds of thousands of soft page faults when this memory is first accessed. By using internal address trees to map the segments holding intermediate map results, MapReduce can benefit from both separate page tables (no sharing costs in the kernel for the address space that is not shared) and shared page tables (avoiding the cost of soft page faults when parts of the address space are truly shared). Section 7 shows that the use of address trees reduces the time that Phoenix spends in the operating system.

5.2 Web server applications

Web server applications provide an interesting case study for exploring multicore application and operating system design issues. A Web server application running on a multicore machine requires sharing operating system services, such as the network stack and the file system, between cores. Furthermore, client requests must be handled with care, because it is possible that the application data required to handle requests exceeds the cache capacity of a single core.

The two main functions performed by Web server applications are processing HTTP requests and performing some function based on the HTTP requests—that is, the applications we are targeting do more than just serving static Web pages. The goal of the design is to perform these functions efficiently by utilizing the hardware resources well.

The Corey Web server is built from three components: spatial web daemons (`swebd`), kernel cores, and applications. `Swebd` is responsible for processing HTTP requests. Every core devoted to `swebd` runs a separate network stack and starts a kernel core with a network device and a syscall device. The kernel core is dedicated

to processing incoming and outgoing Ethernet packets. The networking stack on each swebd core feeds receive and transmit buffers asynchronously to the kernel core. Swebd parses HTTP requests and hands them off to the corresponding application. Applications run on several cores; swebd forwards requests, using a shared memory segment for IPC, to the cores most likely to have the necessary data cached locally. The application core performs the required computation and returns the results to swebd. Our implementation uses the Corey buffer cache to store application data, so a request that is forwarded to a core without the application data cached can still be completed, but at a cost to performance. Swebd packages application results in an HTTP response and replies to the client by passing packets asynchronously the kernel core. Figure 3 presents one configuration of the Corey Web server.

All kernel objects necessary for a swebd core to complete a request, such as packet buffer segments, network devices, and shared segments used for IPC are referenced by a private per-swebd core share. Furthermore, most kernel objects, with the exception of the IPC segments, are used only by one core. With this design, once IPC segments have been mapped into the swebd and application address trees, cores process requests without contending over any global kernel data or locks, and scalability is only limited by the application. We will show in Section 7 that this property is crucial for good scalability.

If a request's computation on the data is sufficiently intense and the aggregate working set fits in the caches (e.g., an SQL query operating on a table that fits in the L1/L2), dedicating cores to applications can improve throughput at the server, as shown in Section 7, because it will induce fewer remote L1/L2 cache references and it has fewer fine-grained locks (if only a single core accesses the data, no lock is necessary to protect that data).

6 IMPLEMENTATION

Corey runs on x86-64 processors, such as AMD Opteron and Intel Xeon CPUs. Our implementation is simplified by using a 64-bit virtual address space, but nothing in the Corey design relies on a large virtual address space. The implementation of address trees is geared towards architectures that use page tables. Address trees could be ported to TLB-only architectures, such as MIPS, but would provide less performance benefit, because every core must fill its own TLB.

The Corey kernel does not implement traditional system services, such as schedulers and network stacks, so the implementation is only 11,000 lines of C and 150 lines of assembly. About 4,000 lines of code are devoted to kernel services, such as address trees and processor objects. Another 3,000 lines of code are required for architecture specific functions, such as page table manipulations and CPU control. Corey device drivers make up the last 4,000 lines of code.

A user-level library, 10,000 lines of C/C++, provides Corey-specific glue for uClibc and Streamflow and includes commonly used functions, such as cfork.

Swebd and several extensions are implemented in 1,000 lines of C++. Swebd lacks many of the features found in industrial Web servers such as Apache, but it is still capable of serving static files and dynamic content using extensions written in C++. As mentioned in Section 4.3, the network stack is lwIP, and the port requires 1,000 lines of C. A SQL database, SQLite[4], is ported to Corey. The port requires 1,500 lines of C.

Corey includes virtual machine support capable of booting Linux. We will use Linux to run legacy applications. The virtual machine implementation relies on AMD SVM extensions, and we plan to add support for Intel VMX.

7 EVALUATION

This section demonstrates that the Corey approach of avoiding sharing in the kernel by default and exposing the benefits and costs of sharing in the kernel to library operating systems can be beneficial to applications. This section also demonstrates that dedicating cores to functions and data can be beneficial. We make these points using MapReduce and Web server applications, as well as using several microbenchmarks.

7.1 Experimental setup

We compare Corey and Linux running on the AMD 16-core and Intel 16-core systems (see Figure 1 in Section 2). The AMD machine has 64 Gbyte of memory and two network cards. The Intel machine has 32 Gbyte of memory and eight network cards. All Linux experiments use Debian Linux with kernel version 2.6.24.3 and Apache 2.2.8 for Web server benchmarks. We disabled logging in Apache to avoid potential performance problems associated with flushing the log to disk or serializing updates. For the networking benchmarks we use sixteen 3 GHz Pentium 4 client machines to generate requests. Each client machine has 1 Gbyte of memory and one network card. All machines used gigabit Intel e1000 Ethernet adapter cards and are connected by a gigabit switched network.

Corey and swebd are not as complete as Linux and Apache. The lack of features could have both a positive and negative impact on performance. For example, swebd does not support all HTTP features, which might enable faster request parsing. On the other hand, the lack of sendfile support in Corey library operating systems and swebd means that swebd is unable to meet Apache's throughput for large file downloads. This

[4] http://www.sqlite.org/

drawback is compounded by the fact lwIP is not as efficient as Linux's network stack.

For the Corey `swebd` experiments and buffer cache microbenchmarks, the buffer cache was not backed by a disk. Contents were loaded into the buffer cache from memory. We warmed the buffer cache on Linux before taking any measurements by reading each file involved in the experiment.

For the networking benchmarks, we took great care to configure Linux to achieve good performance. We have tried to make performance comparisons fair in our evaluation by disabling Apache features known to degrade performance and enabling features known to improve performance. We compiled the Linux e1000 driver with NAPI [18] enabled, which automatically switches to polling under heavy load. We balanced load across the cores by statically assigning IRQ affinity for each Ethernet adapter to a unique set of cores. All Ethernet adapters are connected to the same subnet, which can cause load imbalance if the Linux network stack uses the default routing and ARP policies. We prevented load imbalance by manually configuring server routing and client ARP tables. These configuration changes would be impractical, if not impossible, in practice, so the Linux results are truly best case ones.

We gathered detailed cache statistics using the hardware event registers present on Intel and AMD CPUs. The registers count hardware events such as the number of L1 cache misses. On Linux, we used OProfile[5] to access the hardware event registers. Instead of counting hardware events directly, OProfile programs the hardware event registers to raise an interrupt when they overflow. The initial values of the hardware counters and the overhead of handling an interrupt may affect absolute numbers, but we believe the magnitudes and trends are sufficient for our purposes. We measured hardware events in Corey using the hardware event counters directly. All experiments were run twice: once without profiling to measure execution time and once with profiling to measure cache usage.

7.2 The benefits and costs of sharing

Using the Corey port of the optimized Phoenix MapReduce library we demonstrate that exposing the benefits and costs of sharing in the kernel matters to application performance. As described in Section 4, Phoenix uses internal address trees to share only the page-table entries that map memory holding the intermediate results produced by the map phase and consumed by the reduce phase. Our results show that internal address trees are an important factor in scaling Phoenix on Corey and outperforming Linux.

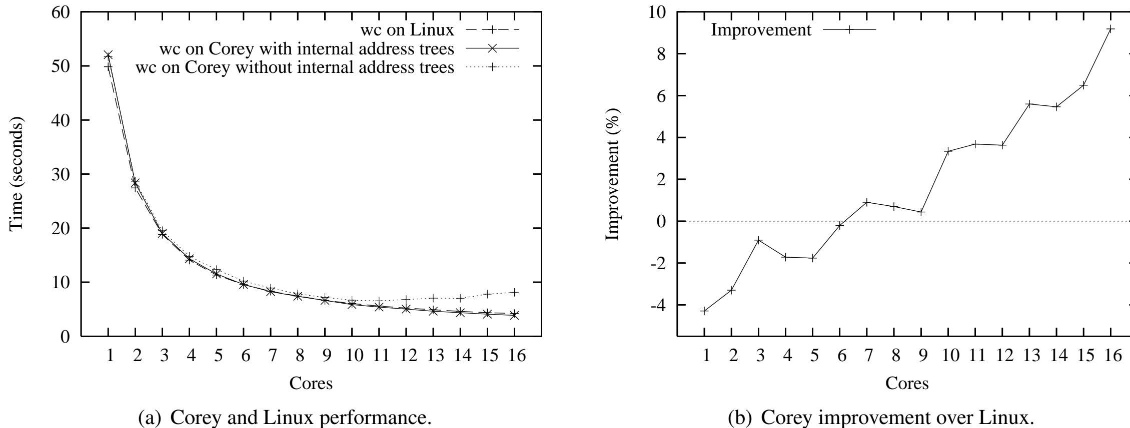
[5] `http://oprofile.sourceforge.net/`

To see the benefit of the Corey approach we compare the performance of optimized Phoenix on Corey (with and without internal address trees) and Linux. We chose the wordcount (`wc`) MapReduce application, which for this experiment counts the number of occurrences of each word in a 1 Gbyte input file. We believe `wc`'s interaction with the operating system is representative of MapReduce applications that produce a large number of intermediate results. The optimized Phoenix runtime allocates more than 100 Mbyte to hold intermediate results (unoptimized Phoenix allocates more than 1 Gbyte) and takes roughly 30,000 soft page faults in Linux and in Corey when using internal address trees. We optimized the Phoenix runtime to minimize interactions with the virtual memory system, but the virtual memory system's implementation still has a significant impact on performance. We experimented with other MapReduce applications that interacted with the operating system very little, such as matrix multiplication. As expected, performance of these applications was similar on Corey and Linux, and we omit them from our discussion. We ensured that the input file was loaded into memory before taking performance measurements.

Figure 4(a) shows that `wc` on Corey with internal address trees outperforms `wc` on Linux. Corey performs worse than Linux for fewer than 4 cores, because Corey's page fault handler is not optimized. However, Figure 4(b) shows that Corey's performance improves relative to Linux for `wc` with more cores, with maximum improvement of about 9.2% speedup on 16 cores (3.87s versus 4.22s). A breakdown of the performance shows that the biggest relative speedup comes from the reduce phase, which achieves 24% performance speedup (0.22s versus 0.29s). The improvement on 16 cores for the map phase is about 7.7% (3.59s versus 3.86s), and the speedup is about 8.4% for the merge phase.

To understand the performance difference between Corey with internal address trees and Linux, we profiled `wc` on both Linux and Corey on 16 cores. The results from OProfile show that `wc` on Linux spends 11% of the map phase and 17% of the reduce phase in `smp_invalidate_interrupt` on 16 cores. That function is responsible for flushing TLB entries when memory is unmapped from an address space shared by several cores. The Phoenix runtime for Linux runs all threads in one address space. At various execution stages `libpthread` reclaims thread stack space by unmapping stack pages, causing TLB shootdowns. Corey reclaims stack pages too, but avoids remote TLB shootdowns because each core maps the stack of its currently running thread into a per-core root address tree.

Changing the semantics of `munmap` to batch unmap operations would decrease the number of TLB shootdowns without needing address trees. However, this may



(a) Corey and Linux performance.

(b) Corey improvement over Linux.

Figure 4: MapReduce `wc` results.

not be feasible if memory or virtual address space are not abundant, and it still does not eliminate remote TLB shootdowns, which cost thousands of cycles for both the senders and receivers.

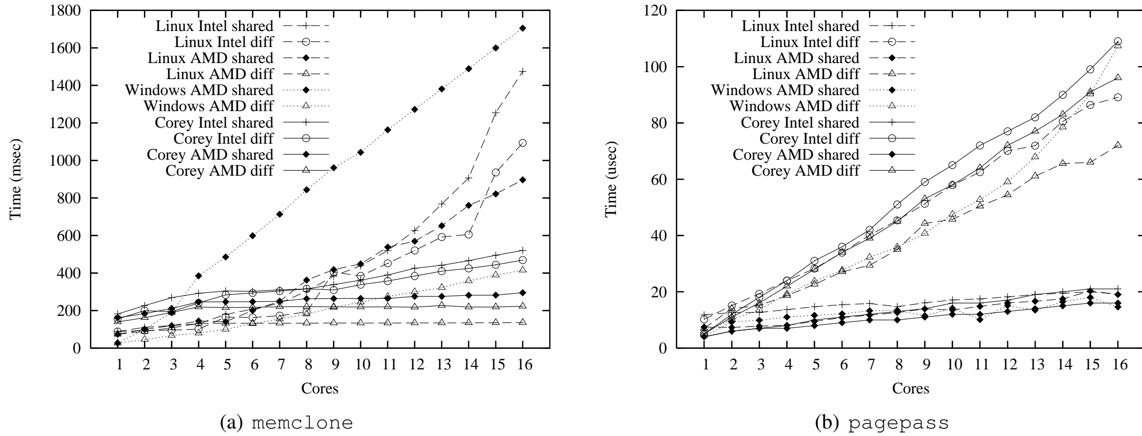
To understand the performance benefits of address trees, we also present the performance data of `wc` without utilizing address trees. As shown in Figure 4(a), `wc` without address tree scales badly. The performance slowdown is about 110% compared to that with address tree optimization (8.13s versus 3.87s). The performance slowdown comes from the increased number of soft page faults in the reduce phase. In this phase, each core needs to fetch the data from other cores, so not sharing the page entries increases the number of soft page faults by 15 times on 16 cores. For `wc` running on 16 cores, the time spent on the reduce phase increases from 0.22s to 4.19s if address trees are not used. The time spent on the merge phase also increases noticeably (from 0.06s to 0.12s).

To isolate the benefits of address trees, we ran two microbenchmarks: `memclone` [3] and `pagepass`. In `memclone`, each core allocates its own 100 Mbyte array in an address space shared across all cores. After allocating, each core touches each page in the array. The benchmark measures the time for the last thread to complete. Figure 5(a) presents the results. For the Corey shared address space experiments, each core has a private root address tree and a shared interior address tree. All cores map all shared interior address trees and cores map the 100 Mbyte segments into their own shared interior address tree. As expected, the shared address space scales well on Corey but poorly on other systems because of contention on shared data structures in the kernel. OProfile results on the AMD machine show that the functions `__down_read_trylock` and `__up_read`, which protect the shared `mm_struct` with the read-write semaphore `mmap_sem`, take 34.0% and 37.5% of the execution time, respectively. On the Intel machine these two functions take 36.2% and 34.6% of the execution time, respectively.

For the separate address space case, each `memclone` process runs on a core with its own address space. An ideal kernel should be able to complete `memclone` on one core in the same amount of time as `memclone` on 16 cores. Linux on the Intel machine provides poor scalability. Results from OProfile show that 54.8% of the execution time is in the function `rmqueue_bulk`, which must acquire the spin lock of the `zone` structure. Linux on AMD performs better because it has four zonelists (and has some architectural advantages); however, `memclone` on 16 cores still takes twice as long as `memclone` on one core. This behavior is not specific to Linux; the figure shows similar results on Windows.

The experiments with `memclone` show that there is a cost in sharing an address space, but sharing can also be beneficial, as demonstrated by MapReduce. We measure this benefit in isolation using the `pagepass` microbenchmark. This benchmark allocates a 4 KB shared buffer on one core and touches it, and then passes it to another core, which repeats the process until all cores have touched the buffer. `Pagepass` represents a worst-case scenario for separate address spaces. Figure 5(b) presents the results and shows that a design with separate address spaces is costly under this workload, because each core takes a soft page fault. (The first page fault is more expensive than subsequent page faults because the backing page for the buffer must be allocated and cleared.)

Address trees allow applications to balance the benefits and costs of sharing page table entries. This flexibility can allow them to achieve better performance, as demonstrated by the MapReduce applications.



(a) `memclone` (b) `pagepass`

Figure 5: Virtual memory management microbenchmark results.

7.3 Dedicating cores to functions and data

The large number of cores and the importance of using caches well should make it attractive to dedicate cores to running specific functions or storing specific data. The benefits of good cache usage and not having to use locks should outweigh the cost of "wasting" cores on dedicated tasks. Using a synthetic Web server application we demonstrate that even on today's machines with a small number of cores such benefits can be indeed obtained. We use a synthetic application, so that we can explore when the benefits outweigh the costs.

The synthetic application is a simple `filesum` application. `Filesum` parses HTTP GETs, opens the requested file, sums the contents and returns the result. `Filesum` can be configured to run in *conventional* or *locality* mode. In conventional mode, the core receiving the request reads the requested file from the buffer cache, sums the file's content, and sends the response. When `filesum` is running in locality mode, the Web server statically assigns files to `filesum` cores, and the request is forwarded to the core assigned the requested file; that core sums the content of the file, which requires no remote memory accesses.

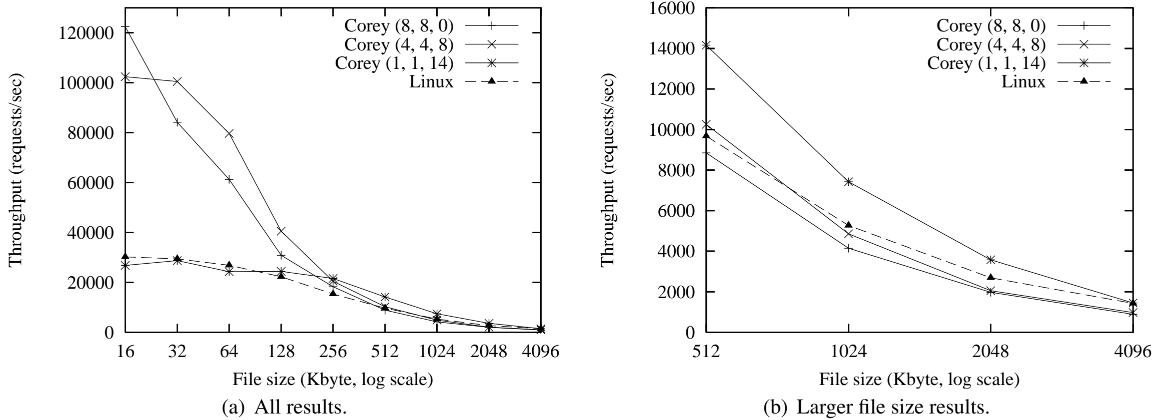
We have implemented the `filesum` application on Linux and Corey. It is written in C and runs on Corey as an extension to `swebd` and on Linux as an Apache module. The `filesum` application avoids pathname resolution for each request by caching file descriptors, and the Apache module handles GETs without checking a `.htaccess` file. Apache `filesum` does not support locality mode, because we are unaware of any mechanisms for reserving application cores using the Apache module interface.

Figure 6 presents the throughput of `filesum` under three Corey configurations and one Linux configuration. The Corey configurations are composed of the three components described in Section 5.2: `swebd`, kernel cores, and applications. In the configurations presented, each kernel core manages one Ethernet adapter and executes system calls invoked asynchronously by a `swebd` core. After a `swebd` core parses a `filesum` request, it forwards the request to the `filesum` application core responsible for the file. The configuration that achieves the highest throughput depends on how much data must be processed to complete a request.

When `filesum` sums only small files, `swebd` and the network stack are bottlenecks. In this case, it is most efficient to use conventional mode, where `swebd` and the application execute on the same cores and all data is replicated in the hardware caches. When the size of the application's working set approaches the cache capacity of a single core, locality mode is better; it is able to complete 50% more requests per second than the conventional mode on both Corey and Linux. As the application working set grows larger than the aggregate size of application cores, the performance of `filesum` in locality mode is dominated by cache misses and approaches that of `filesum` in conventional mode.

To explore these issues in the context of a more realistic workload we used the SQLite library to implement a database join `swebd` extension. The extension joins a set of private tables with a second set of globally-shared tables. Like `filesum`, it uses locality mode. The private tables are pinned to a core and stored in a core-private heap. The queries are routed to their assigned core by the `swebd` front-ends. The globally-shared tables are mapped into a shared address region using Corey's interior address trees.

Figure 7 shows the results, comparing the `swebd` SQLite extension with an Apache SQLite module. The pattern is similar to that observed in Figure 6. At low table sizes, Corey benefits from locality. As table sizes increase beyond the hardware caches, locality becomes a less dominant factor.



(a) All results.

(b) Larger file size results.

Figure 6: Throughput of `filesum` under three Corey configurations and one Linux configuration. The Corey keys in the legend indicate the configuration used. The first element in the list is the number of cores running `swebd` and a network stack, the second is the number of kernel cores, and the third element is the number of cores devoted to `filesum`. The configuration with 14 application cores stripes file contents across two cores when the file size is greater than or equal to 1024 Kbyte. The clients randomly selected one of eight files for each request.

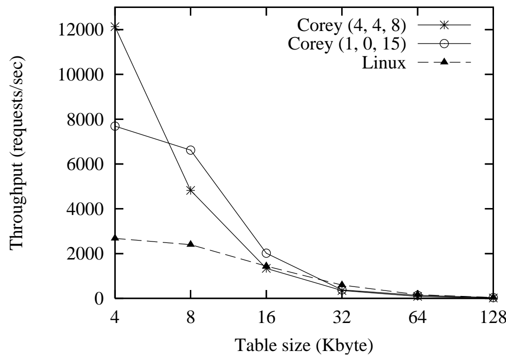


Figure 7: Web server with SQLite database, join queries. The Corey keys in the legend describe the configuration as they do in Figure 6.

The `filesum` application in locality mode assigns data to cores as well as certain library operating system functions to cores. To isolate the benefits from assigning library operating system functions to cores, we measured the performance of network workloads with many small short network connections (as in `filesum`) on Corey and Linux using a simple TCP benchmark.

A TCP server accepts an incoming connection, writes a 128 byte response, and closes the connection. We configure Corey and Linux to achieve the highest throughput as follows. Corey runs two cores per Ethernet adapter. One core runs a networking stack and the simple TCP server, and the other core is a kernel core, executing system calls from the first core and polling the Ethernet adapter. Similar to `swebd` cores, TCP server cores reference all kernel objects necessary to complete a connection using a private share. Under Linux every core runs a simple user-level TCP server with two cores on the same die sharing one Ethernet adapter. We also implemented an in-kernel TCP server using Linux syslets.[6] Its performance was the same as the user-level TCP server and it suffered from the same bottlenecks, so we omit the results.

Figure 8 presents the results of the TCP microbenchmark. Corey scales almost perfectly, while Linux performance grows more slowly and eventually begins to degrade. Linux scalability is limited by two global spin locks the kernel acquires when creating and releasing a socket. The Linux VFS code acquires both the `inode_lock` and `dcache_lock` to add and remove socket metadata from global lists and queues, even though sockets are only ever used by one core.

Corey provides better scalability than Linux for two reasons. First, Corey shares allow different pairs of cores to complete connections without contending on global locks or data structures in the kernel or library operating system. Second, the kernel core efficiently handles the Ethernet adapter for each pair of cores and allows the TCP server to invoke system calls asynchronously. In Corey, if a TCP server starts a kernel core, throughput improves by a factor of 2. In Linux if a TCP server forks another process to use the same Ethernet adapter performance improves by only a factor of 1.7 in the best case.

A final factor in the performance of `filesum` is the buffer cache implementation. For small files the buffer cache is a limiting factor in performance for Linux, but not for Corey. To isolate the effects of the buffer cache, we wrote a read microbenchmark that performs 100,000 reads per core of a 4 KB-sized file using `pread`. We compiled Linux without `dnotify` and `inotify` support, because the file access notification code can con-

[6]Linux syslets are a patch for the mainline kernel and bear some resemblance to Corey kernel cores but were designed for a different purpose and provide a much different interface.

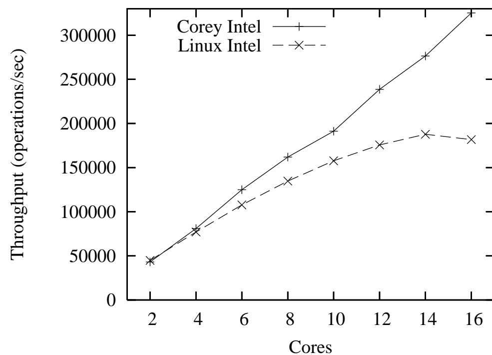


Figure 8: TCP microbenchmark results for Corey and Linux.

sume a large number of cycles, even if no daemons are listening for events.

On the 16-core Intel machine Corey completes 1.7 million reads per second on one core and scales linearly to complete 27.2 million reads per second on 16 cores. Linux completes 1.2 million reads per second on one core and degrades to 1.1 million reads per second on 16 cores. The main reason is that Linux incurs cache misses to acquire a read-write lock protecting the buffer cache, while Corey avoids these using the special lock described in Section 4.2. Results from OProfile show that Linux incurs approximately ten remote cache misses per read operation, while Corey incurs no remote cache misses per read operation. We implemented an experimental Linux kernel patch that replaces the standard radix tree read-write lock with the special Corey read-write lock. For 16 cores, throughput of the read microbenchmark more than doubles, but further improvement is constrained by additional bottlenecks that we have not investigated.

8 DISCUSSION AND FUTURE WORK

The results above should be viewed as a case for Corey's design rather than a conclusive "proof". Many of the features implemented in Linux are not in Corey, which influences the results both positively and negatively. Corey lacks a system-wide scheduler for when the applications together require more cores than available. Corey does not provide security features, but we believe that shares are a good abstraction that can easily incorporate HiStar's information flow control [26]. Corey also doesn't provide a good file system yet, but we believe that Corey's buffer cache will fit well with a striping file system. It is also likely that some of the Corey abstractions could be implemented in Linux and provide benefits to Linux too. We intend to explore these topics in future work.

Because of its organization, Corey is a convenient testbed for rapidly exploring different designs to address other challenges in multicore systems: scaling frequency to exploit peak performance while controlling temperature; tolerating core failure [7] but exploiting application semantics (e.g., MapReduce applications can already handle failures); scalable virtual machines, etc. We also intend to explore other applications and continue to work on the current applications (e.g., running stages of a Web server like SEDA [24] on different cores).

9 RELATED WORK

Corey is related to research in NUMA operating systems and multicore. We discuss the relationships in turn.

9.1 NUMA operating systems

Multicore systems exhibit properties similar to those of NUMA machines: access to local memory is fast, while remote memory is slow. The main difference is that more cores (and their caches) are put on a single processor, adding another dimension to locality considerations. Because multicore systems are similar to NUMA systems, research on NUMA operating systems is relevant to Corey.

Corey is influenced by Disco [6] (and its relatives [13, 23]), which runs on a NUMA machine, but its kernel interface is like an exokernel [10] rather than a virtual machine monitor. Like Disco, Corey organizes the operating system more like a distributed operating system than an SMP operating system, tries to avoid making the kernel a performance bottleneck (e.g., by avoiding locks and using an optimized buffer cache), and aims for a small kernel to ease optimizations.

Corey focuses on supporting library operating systems for multicore applications, rather than virtualization. This bears some resemblance to Disco's SPLASHOS, a runtime tailored for running the Splash benchmark [20], but Corey's library operating systems require more operating system features. To support them well, the Corey kernel provides a number of novel ideas to allow them to control sharing (address trees, shares, and kernel cores).

K42 [3] and Tornado [11], like Corey, are designed so that independent applications do not contend over resources managed by the kernel. The K42 and Tornado kernels allow different implementations of objects based on sharing patterns. The Corey kernel takes a different approach; library operating systems control sharing so that they can implement higher-level operating systems abstractions that fit best the sharing requirements for the target application domain. In addition, Corey provides new abstractions not present in K42 and Tornado.

Much work has been done on making widely-used operating systems run well on NUMA systems [5]. The Linux community[7] has introduced many changes to Linux for NUMA scalability (e.g., Read-Copy-Update

[7]The Linux symposium (`http://www.linuxsymposium.org`) features papers related to NUMA almost every year

(RCU) [15] locks, a scheduler with local runqueues [1], `libnuma` [14], a rewrite of load-balancing support[8]). These techniques are complementary to the new techniques that Corey introduces, but have been a source of inspiration. For example, Corey uses tricks inspired by RCU to implement efficient functions for retrieving and removing objects from a share.

9.2 Multicore research

Saha et al. have proposed the Multi-Core Run Time (McRT) for emerging desktop workloads [17] and explored configurations in which McRT runs on the bare metal, with an operating system running on separate cores. Corey also uses spatial multiplexing, but provides a new kernel that runs on all cores and that allows applications to dedicate kernel functions to cores through kernel cores.

Several studies with Linux on multicore processors [12, 22] have identified challenges in scaling existing operating systems to many cores, which inspired us to work on Corey.

Recently, new optimizations, such as thread clustering [21] and constructive cache sharing [8], have been proposed; these optimizations are orthogonal to Corey's design.

10 CONCLUSIONS

In order for applications to scale on modern multicore architectures, this paper argues operating systems must yield sharing to application control. Corey is a new kernel that follows this principle. Its novel share, address tree, and kernel core abstractions ensure that all kernel data structures are local to a core by default, while giving applications the flexibility to share these structures only when necessary. Experiments with a MapReduce application and a synthetic Web application demonstrate that Corey's design allows it to avoid traditional bottlenecks and outperform Linux for these applications on available 16-core machines.

REFERENCES

[1] J. Aas. Understanding the Linux 2.6.8.1 CPU scheduler, February 2005. http://josh.trancesoftware.com/linux/.

[2] A. Agarwal and M. Levy. Thousand-core chips: the kill rule for multicore. In *Proceedings of the 44th Annual Conference on Design Automation*, pages 750–753, 2007.

[3] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience distributing objects in an SMMP OS. *ACM Trans. Comput. Syst.*, 25(3):6, 2007.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, USA, October 2003. ACM.

[5] R. Bryant, J. Hawkes, J. Steiner, J. Barnes, and J. Higdon. Scaling linux to the extreme. In *Proceedings of the Linux Symposium 2004*, pages 133–148, Ottawa, Ontario, June 2004.

[6] E. Bugnion, S. Devine, and M. Rosenblum. DISCO: running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 143–156, Saint-Malo, France, October 1997. ACM.

[7] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 12–25, Copper Mountain, CO, December 1995. ACM.

[8] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling Threads for Constructive Cache Sharing on CMPs. In *Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115. ACM, 2007.

[9] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[10] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, December 1995. ACM.

[11] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 87–100, 1999.

[12] C. Gough, S. Siddha, and K. Chen. Kernel scalability—expanding the horizon beyond fine grain locks. In *Proceedings of the Linux Symposium 2007*, pages 153–165, Ottawa, Ontario, June 2007.

[13] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 154–169, Kiawah Island, SC, October 1999. ACM.

[14] A. Klein. An NUMA API for Linux, August 2004. http://www.firstfloor.org/~andi/numa.html.

[15] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read-copy update. In *Proceedings of the Linux Symposium 2002*, pages 338–367, Ottawa, Ontario, June 2002.

[16] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE Computer Society, 2007.

[17] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling scalability and performance in a large-scale CMP environment. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 73–86, New York, NY, USA, 2007. ACM.

[18] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Proceedings of the 5th annual conference on Linux Showcase & Conference*, pages 18–18, Berkeley, CA, USA, 2001. USENIX Association.

[19] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable Locality-Conscious Multithreaded Memory Allocation. In *Proceedings of the 2006 ACM SIGPLAN International Symposium on Memory Management*, pages 84–94, 2006.

[20] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44.

[21] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 47–58, New York, NY, USA, 2007. ACM.

[22] B. Veal and A. Foong. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 57–66, New York, NY, USA, 2007. ACM.

[23] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the 7th international conference on Architectural support for programming languages and operating systems*, pages 279–289, New York, NY, USA, 1996. ACM.

[24] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 230–243, Banff, Canada, October 2001. ACM.

[25] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. *SIGMETRICS Perform. Eval. Rev.*, 33(1):181–192, 2005.

[26] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, November 2006.

[8] http://kerneltrap.org/node/8059