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 application performance can be limited by the operating system when the application uses the operating system frequently and the operating system services use data structures shared and modified by multiple processing cores. If the application does not need the sharing, then the operating system will become an unnecessary bottleneck to the application's performance.

This paper argues that *applications should control sharing*: the kernel should arrange each data structure so that only a single processor need update it, unless directed otherwise by the application. Guided by this design principle, this paper proposes three operating system abstractions (address ranges, kernel cores, and shares) that allow applications to control inter-core sharing and to take advantage of the likely abundance of cores by dedicating cores to specific operating system functions.

Measurements of microbenchmarks on the Corey prototype operating system, which embodies the new abstractions, show how control over sharing can improve performance. Application benchmarks, using MapReduce and a Web server, show that the improvements can be significant for overall performance: MapReduce on Corey performs 25% faster than on Linux when using 16 cores. Hardware event counters confirm that these improvements are due to avoiding operations that are expensive on multicore machines.

1 INTRODUCTION

Cache-coherent shared-memory multiprocessor hardware has become the default in modern PCs as chip manufacturers have adopted multicore architectures. Chips with four cores are common, and trends suggest that chips with tens to hundreds of cores will appear within five years [2]. This paper explores new operating system abstractions that allow applications to avoid bottlenecks in the operating system as the number of cores increases.

Operating system services whose performance scales poorly with the number of cores can dominate application performance. 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 as a Linux Web server uses more cores directory lookups spend increasing amounts of time contending for spin locks [29]. Section 8.5.1 shows that contention for Linux address space data structures causes the percentage of total time spent in the reduce phase of a MapReduce application to increase from 5% at seven cores to almost 30% at 16 cores.

One source of poorly scaling operating system services is use of data structures modified by multiple cores. Figure 1 illustrates such a scalability problem with a simple microbenchmark. The benchmark creates a number of threads within a process, each thread creates a file descriptor, and then each thread repeatedly duplicates (with `dup`) its file descriptor and closes the result. The graph shows results on a machine with four quad-core AMD Opteron chips running Linux 2.6.25. Figure 1 shows that, as the number of cores increases, the total number of `dup` and `close` operations per unit time *decreases*. The cause is contention over shared data: the table describing the process's open files. With one core there are no cache misses, and the benchmark is fast; with two cores, the cache coherence protocol forces a few cache misses per iteration to exchange the lock and table data. More generally, only one thread at a time can update the shared file descriptor table (which prevents any increase in performance), and the increasing number of threads spinning for the lock gradually increases locking costs. This problem is not specific to Linux, but is due to POSIX semantics, which require that a new file descriptor be visible to all of a process's threads even if only one thread uses it.

Common approaches to increasing scalability include avoiding shared data structures altogether, or designing them to allow concurrent access via fine-grained locking or wait-free primitives. For example, the Linux community has made tremendous progress with these approaches [18].

A different approach exploits the fact that some instances of a given resource type need to be shared, while others do not. If the operating system were aware of an application's sharing requirements, it could choose resource implementations suited to those requirements. A

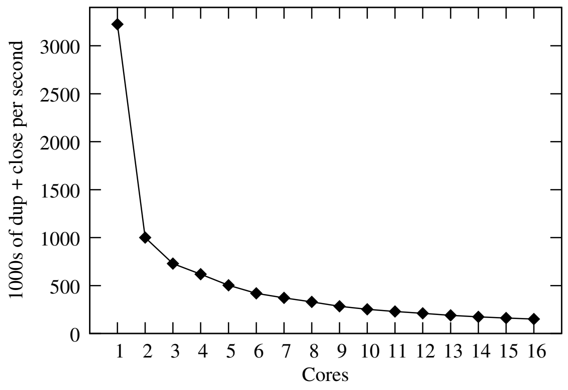


Figure 1: Throughput of the file descriptor `dup` and `close` microbenchmark on Linux.

limiting factor in this approach is that the operating system interface often does not convey the needed information about sharing requirements. In the file descriptor example above, it would be helpful if the thread could indicate whether the new descriptor is private or usable by other threads. In the former case it could be created in a per-thread table without contention.

This paper is guided by a principle that generalizes the above observation: applications should control sharing of operating system data structures. "Application" is meant in a broad sense: the entity that has enough information to make the sharing decision, whether that be an operating system service, an application-level library, or a traditional user-level application. This principle has two implications. First, the operating system should arrange each of its data structures so that by default only one core needs to use it, to avoid forcing unwanted sharing. Second, the operating system interfaces should let callers control how the operating system shares data structures between cores. The intended result is that the operating system incurs sharing costs (e.g., cache misses due to memory coherence) only when the application designer has decided that is the best plan.

This paper introduces three new abstractions for applications to control sharing within the operating system. *Address ranges* allow applications to control which parts of the address space are private per core and which are shared; manipulating private regions involves no contention or inter-core TLB invalidations, while explicit indication of shared regions allows sharing of hardware page tables and consequent minimization of soft page faults (page faults that occur when a core references pages with no mappings in the hardware page table but which are present in physical memory). *Kernel cores* allow applications to dedicate cores to run specific kernel functions, avoiding contention over the data those functions use. *Shares* are lookup tables for kernel objects that allow applications to control which object identifiers are visible to other cores. These abstractions are implementable without inter-core sharing by default, but allow sharing among cores as directed by applications.

We have implemented these abstractions in a prototype operating system called Corey. Corey is organized like an exokernel [9] to ease user-space prototyping of experimental mechanisms. The Corey kernel provides the above three abstractions. Most higher-level services are implemented as library operating systems that use the abstractions to control sharing. Corey runs on machines with AMD Opteron and Intel Xeon processors.

Several benchmarks demonstrate the benefits of the new abstractions. For example, a MapReduce application scales better with address ranges on Corey than on Linux. A benchmark that creates and closes many short TCP connections shows that Corey can saturate the network device with five cores by dedicating a kernel core to manipulating the device driver state, while 11 cores are required when not using a dedicated kernel core. A synthetic Web benchmark shows that Web applications can also benefit from dedicating cores to data.

This paper should be viewed as making a case for controlling 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, changes in the architecture of future multicore processors may change the attractiveness of the ideas presented here.

The rest of the paper is organized as follows. Using architecture-level microbenchmarks, Section 2 measures the cost of sharing on multicore processors. Section 3 describes the three proposed abstractions to control sharing. Section 4 presents the Corey kernel and Section 5 its operating system services. Section 6 summarizes the extensions to the default system services that implement MapReduce and Web server applications efficiently. Section 7 summarizes the implementation of Corey. Section 8 presents performance results. Section 9 reflects on our results so far and outlines directions for future work. Section 10 relates Corey to previous work. Section 11 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 on the interconnect.

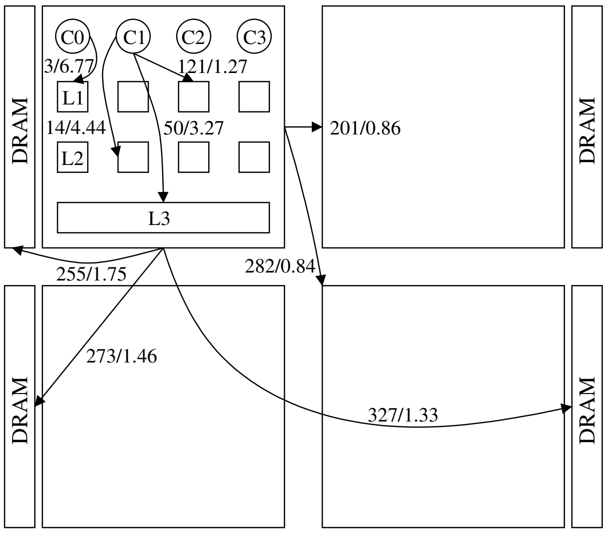


Figure 2: The AMD 16-core system topology. Memory access latency is in cycles and listed before the backslash. Memory bandwidth is in bytes per cycle and listed after the backslash. The measurements reflect the latency and bandwidth achieved by a core issuing load instructions. The measurements for accessing the L1 or L2 caches of a different core on the same chip are the same. The measurements for accessing any cache on a different chip are the same. Each cache line is 64 bytes, L1 caches are 64 Kbytes 8-way set associative, L2 caches are 512 Kbytes 16-way set associative, and L3 caches are 2 Mbytes 32-way set associative.

These properties can already be observed in current multicore machines. Figure 2 summarizes the memory system of a 16-core machine from AMD. The machine has four quad-core Opteron processors connected by a square interconnect. The interconnect carries data between cores and memory, as well as cache coherence broadcasts to locate and invalidate cache lines, and point-to-point cache coherence transfers of individual cache lines. Each core has a private L1 and L2 cache. Four cores on the same chip share an L3 cache. Cores on one chip are connected by an internal crossbar switch and can access each others' L1 and L2 caches efficiently. Memory is partitioned in four banks, each connected to one of the four chips. Each core has a clock frequency of 2.0 GHz.

Figure 2 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. [31]. 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.

Figure 3 shows the performance scalability of locking, an important primitive often dominated by cache miss costs. The graph compares Linux's kernel spin locks and

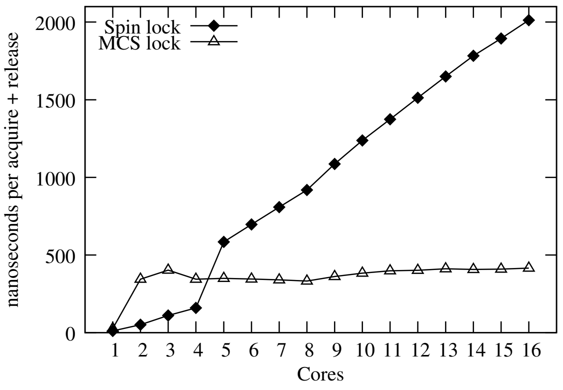
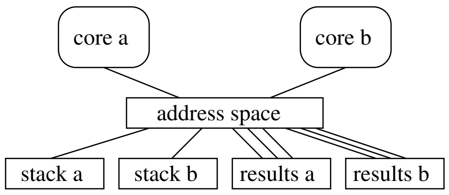


Figure 3: Time required to acquire and release a lock on a 16-core AMD machine when varying number of cores contend for the lock. The two lines show Linux kernel spin locks and MCS locks (on Corey). A spin lock with one core takes about 11 nanoseconds; an MCS lock about 26 nanoseconds.

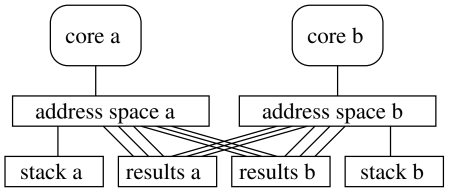
scalable MCS locks [21] (on Corey) on the AMD machine, varying the number of cores contending for the lock. For both the kernel spin lock and the MCS lock we measured the time required to acquire and release a single lock. When only one core uses a lock, both types of lock are fast because the lock is always in the core's cache; spin locks are faster than MCS locks because the former requires three instructions to acquire and release when not contended, while the latter requires 15. The time for each successful acquire increases for the spin lock with additional cores because each new core adds a few more cache coherence broadcasts per acquire, in order for the new core to observe the effects of other cores' acquires and releases. MCS locks avoid this cost by having each core spin on a separate location.

We measured the average times required by Linux 2.6.25 to flush the TLBs of all cores after a change to a shared page table. These "TLB shootdowns" are considerably slower on 16 cores than on 2 (18742 cycles versus 5092 cycles). TLB shootdown typically dominates the cost of removing a mapping from an address space that is shared among multiple cores.

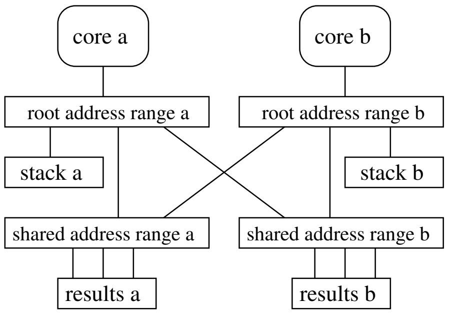
On AMD 16-core systems, the speed difference between a fast memory access and a slow memory access is a factor of 100, and that factor is likely to grow with the number of cores. Our measurements for the Intel Xeon show a similar speed difference. For performance, kernels must mainly access data in the local core's cache. A contended or falsely shared cache line decreases performance because many accesses are to remote caches. Widely-shared and contended locks also decrease performance, even if the critical sections they protect are only a few instructions. These performance effects will be more damaging as the number of cores increases, since the cost of accessing far-away caches increases with the number of cores.



(a) A single address space.



(b) Separate address spaces.



(c) Two address spaces with shared result mappings.

Figure 4: Example address space configurations for MapReduce executing on two cores. Lines represent mappings. In this example a stack is one page and results are three pages.

3 Design

Existing operating system abstractions are often difficult to implement without sharing kernel data among cores, regardless of whether the application needs the shared semantics. The resulting unnecessary sharing and resulting contention can limit application scalability. This section gives three examples of unnecessary sharing and for each example introduces a new abstraction that allows the application to decide if and how to share. The intent is that these abstractions will help applications to scale to large numbers of cores.

3.1 Address ranges

Parallel applications typically use memory in a mixture of two sharing patterns: memory that is used on just one core (private), and memory that multiple cores use (shared). Most operating systems give the application a choice between two overall memory configurations: a single address space shared by all cores or a separate address space per core. The term address space here refers to the description of how virtual addresses map to memory, typically defined by kernel data structures and instantiated lazily (in response to soft page faults) in hardware-defined page tables or TLBs. If an application chooses to harness concurrency using threads, the result is typically a single address space shared by all threads. If an application obtains concurrency by forking multiple processes, the result is typically a private address space per process; the processes can then map shared segments into all address spaces. The problem is that each of these two configurations works well for only one of the sharing patterns, placing applications with a mixture of patterns in a bind.

As an example, consider a MapReduce application [8]. During the map phase, each core reads part of the application's input and produces intermediate results; map on each core writes its intermediate results to a different area of memory. Each map instance adds pages to its address space as it generates intermediate results. During the reduce phase each core reads intermediate results produced by multiple map instances to produce the output.

For MapReduce, a single address-space (see Figure 4(a)) and separate per-core address-spaces (see Figure 4(b)) incur different costs. With a single address space, the map phase causes contention as all cores add mappings to the kernel's address space data structures. On the other hand, a single address space is efficient for reduce because once any core inserts a mapping into the underlying hardware page table, all cores can use the mapping without soft page faults. With separate address spaces, the map phase sees no contention while adding mappings to the per-core address spaces. However, the reduce phase will incur a soft page fault per core per page of accessed intermediate results. Neither memory configuration works well for the entire application.

We propose address ranges to give applications high performance for both private and shared memory (see Figure 4(c)). An address range is a kernel-provided abstraction that corresponds to a range of virtual-to-physical mappings. An application can allocate address ranges, insert mappings into them, and place an address range at a desired spot in the address space. If multiple cores' address spaces incorporate the same address range, then they will share the corresponding pieces of hardware page tables, and see mappings inserted by each others' soft page faults. A core can update the mappings in a non-shared address range without contention. Even when shared, the address range is the unit of locking; if only one core manipulates the mappings in a shared address range, there will be no contention. Finally, deletion of mappings from a non-shared address range does not require TLB shootdowns.

Address ranges allow applications to selectively share parts of address spaces, instead of being forced to make an all-or-nothing decision. For example, the MapReduce runtime can set up address spaces as shown in 4(c). Each core has a private root address range that maps all private memory segments used for stacks and temporary objects and several shared address ranges mapped by all other cores. A core can manipulate mappings in its private address ranges without contention or TLB shootdowns. If each core uses a different shared address range to store the intermediate output of its map phase (as shown Figure 4(c)), the map phases do not contend when adding mappings. During the reduce phase there are no soft page faults when accessing shared segments, since all cores share the corresponding parts of the hardware page tables.

3.2 Kernel cores

In most operating systems, when application code on a core invokes a system call, the same core executes the kernel code for the call. If the system call uses shared kernel data structures, it acquires locks and fetches relevant cache lines from the last core to use the data. The cache line fetches and lock acquisitions are costly if many cores use the same shared kernel data. If the shared data is large, it may end up replicated in many caches, potentially reducing total effective cache space and increasing DRAM references.

We propose a kernel core abstraction that allows applications to dedicate cores to kernel functions and data. A kernel core can manage hardware devices and execute system calls sent from other cores. For example, a Web service application may dedicate a core to interacting with the network device instead of having all cores manipulate and contend for driver and device data structures (e.g., transmit and receive DMA descriptors). Multiple application cores then communicate with the kernel core via shared-memory IPC; the application cores exchange packet buffers with the kernel core, and the kernel core manipulates the network hardware to transmit and receive the packets.

This plan reduces the number of cores available to the Web application, but it may improve overall performance by reducing contention for driver data structures and associated locks. Whether a net performance improvement would result is something that the operating system cannot easily predict. Corey provides the kernel core abstraction so that applications can make the decision.

3.3 Shares

Many kernel operations involve looking up identifiers in tables to yield a pointer to the relevant kernel data structure; file descriptors and process IDs are examples of such identifiers. Use of these tables can be costly when multiple cores contend for locks on the tables and on the table entries themselves.

For each kind of lookup, the operating system interface designer or implementer typically decides the scope of sharing of the corresponding identifiers and tables. For example, Unix file descriptors are shared among the threads of a process. Process identifiers, on the other hand, typically have global significance. Common implementations use per-process and global lookup tables, respectively. If a particular identifier used by an application needs only limited scope, but the operating system implementation uses a more global lookup scheme, the result may be needless contention over the lookup data structures.

We propose a share abstraction that allows applications to dynamically create lookup tables and determine how these tables are shared. Each of an application's cores starts with one share (its root share), which is private to that core. If two cores want to share a share, they create a share and add the share's ID to their private root share (or to a share reachable from their root share). A root share doesn't use a lock because it is private, but a shared share does. An application can decide for each new kernel object (including a new share) which share will hold the identifier.

Inside the kernel, a share maps application-visible identifiers to kernel data pointers. The shares reachable from a core's root share define which identifiers the core can use. Contention may arise when two cores manipulate the same share, but applications can avoid such contention by placing identifiers with limited sharing scope in shares that are only reachable on a subset of the cores.

For example, shares could be used to implement file-descriptor-like references to kernel objects. If only one thread uses a descriptor, it can place the descriptor in its core's private root share. If the descriptor is shared between two threads, these two threads can create a share that holds the descriptor. If the descriptor is shared among all threads of a process, the file descriptor can be put in a per-process share. The advantage of shares is that the application can limit sharing of lookup tables and avoid unnecessary contention if a kernel object is not shared. The downside is that an application must often keep track of the share in which it has placed each identifier.

4 COREY KERNEL

Corey provides a kernel interface organized around five types of low-level objects: shares, segments, address ranges, pcores, and devices. Library operating systems provide higher-level system services that are implemented on top of these five types. Applications implement domain specific optimizations using the low-level interface; for example, using pcores and devices they can implement kernel cores. Figure 5 provides an overview of the Corey low-level interface. The following sections describe the design of the five types of Corey kernel objects and how they allow applications to control sharing. Section 5 describes Corey's higher-level system services.

System call	Description
name obj_get_name(obj)	return the name of an object
shareid share_alloc(shareid, name, memid)	allocate a share object
void share_addobj(shareid, obj)	add a reference to a shared object to the specified share
void share_delobj(obj)	remove an object from a share, decrementing its reference count
void self_drop(shareid)	drop current core's reference to a share
segid segment_alloc(shareid, name, memid)	allocate physical memory and return a segment object for it
segid segment_copy(shareid, seg, name, mode)	copy a segment, optionally with copy-on-write or -read
nbytes segment_get_nbytes(seg)	get the size of a segment
void segment_set_nbytes(seg, nbytes)	set the size of a segment
arid ar_alloc(shareid, name, memid)	allocate an address range object
void ar_set_seg(ar, voff, segid, soff, len)	map addresses at voff in ar to a segment's physical pages
void ar_set_ar(ar, voff, ar1, aoff, len)	map addresses at voff in ar to address range ar1
ar_mappings ar_get(ar)	return the address mappings for a given address range
pcoreid pcore_alloc(shareid, name, memid)	allocate a physical core object
pcore pcore_current(void)	return a reference to the object for current pcore
void pcore_run(pcore, context)	run the specified user context
void pcore_add_device(pcore, dev)	specify device list to a kernel core
void pcore_set_interval(pcore, hz)	set the time slice period
void pcore_halt(pcore)	halt the pcore
devid device_alloc(shareid, hwid, memid)	allocate the specified device and return a device object
dev_list device_list(void)	return the list of devices
dev_stat device_stat(dev)	return information about the device
void device_conf(dev, dev_conf)	configure a device
void device_buf(dev, seg, offset, buf_type)	feed a segment to the device object
locality_matrix locality_get(void)	get hardware locality information

Figure 5: Corey system calls. shareid, segid, arid, pcoreid, and devid represent 64-bit object IDs. share, seg, ar, pcore, obj, and dev represent $\langle$share ID, object ID$\rangle$ pairs. hwid represents a unique ID for hardware devices and memid represents a unique ID for per-core free page lists.

4.1 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. If the architecture is NUMA, a core's free page list holds pages from its local memory node. 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.

The Corey kernel generally locks an object's metadata before each use. If the application has arranged things so that the object is only used on one core, the lock and use of the metadata will be fast (see Figure 3), assuming they have not been evicted from the core's cache. Corey uses spin lock and read-write lock implementations borrowed from Linux, its own MCS lock implementation, and a scalable read-write lock implementation inspired by the MCS lock to synchronize access to object metadata.

4.2 Object naming

An application 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. Corey uses shares for this purpose. Applications specify which shares are available on which cores by passing a core's share set to `pcore_run` (see below).

When allocating an object, an application selects a share to hold the object ID. The application uses $\langle$share ID, object ID$\rangle$ pairs to specify objects to system calls. Applications can add a reference to an object in a share with `share_addobj` and remove an object from a share with `share_delobj`. The kernel counts references to each object, freeing the object's memory when the count is zero. By convention, applications maintain a per-core private share and one or more shared shares.

4.3 Memory management

The kernel represents physical memory using the *segment* abstraction. Applications 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; an application can arrange to share a segment between cores by adding it to a share, as described above.

An application uses address ranges to define its address space. Each running core has an associated root address range object containing a table of address mappings. By convention, most applications allocate a root address range for each core to hold core private mappings, such as thread stacks, and use one or more address ranges that are shared by all cores to hold shared segment mappings, such as dynamically allocated buffers. An application uses `ar_set_seg` to cause an address range to map addresses to the physical memory in a segment, and `ar_set_ar` to set up a tree of address range mappings.

4.4 Execution

Corey represents physical cores with *pcore* objects. Once allocated, an application 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 range. A pcore executes until `pcore_halt` is called. 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.

An application 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 set in the context argument. 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 an application to invoke system calls on a kernel core. The application communicates with the *syscall device* via a ring buffer in a shared memory segment.

5 System services

This section describes three system services exported by Corey: execution forking, network access, and a buffer cache. These services together with a C standard library that includes support for file descriptors, dynamic memory allocation, threading, and other common Unix-like features create a scalable and convenient application environment that is used by the applications discussed in Section 6.

5.1 `cfork`

`cfork` is similar to Unix `fork` and is the main abstraction used by applications 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 range as copy-on-write and maps them into the root address range of the new pcore. Callers can instruct `cfork` to share specified segments and address ranges with the new processor. Applications implement fine-grained sharing using shared segment mappings and more coarse-grained sharing using shared address range mappings. `cfork` callers can share kernel objects with the new pcore by passing a set of shares for the new pcore.

5.2 Network

Applications can choose to run several network stacks (possibly one for each core) or a single shared network stack. Corey uses the lwIP [19] networking library. Applications specify a network device for each lwIP stack. If multiple network stacks share a single physical device, Corey virtualizes the network card. Network stacks on different cores that share a physical network device also share the device driver data, such as the transmit descriptor table and receive descriptor table.

All configurations we have experimented with run a separate network stack for each core that requires network access. This design provides good 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).

5.3 Buffer cache

An inter-core shared buffer cache is important to system performance and often necessary for correctness when multiple cores access shared files. Since cores share the buffer cache they might contend on the data structures used to organize cached disk blocks. Furthermore, under write-heavy workloads it is possible that cores will contend for the cached disk blocks.

The Corey buffer cache resembles a traditional Unix buffer cache; however, we found three techniques that substantially improve multicore performance. The first is a lock-free tree that allows multiple cores to locate cached blocks without contention. The second is a write scheme that tries to minimize contention on shared data using per-core block allocators and by copying application data into blocks likely to be held in local hardware caches. The third uses a scalable read-write lock to ensure blocks are not freed or reused during reads.

6 Applications

It is unclear how big multicore systems will be used, but parallel computation and network servers are likely application areas. To evaluate Corey in these areas, we implemented a MapReduce system and a Web server with synthetic dynamic content. Both applications are data-parallel but have sharing requirements and thus are not trivially parallelized. This section describes how the two applications use the Corey interface to achieve high performance.

6.1 MapReduce applications

MapReduce is a framework for data-parallel execution of simple programmer-supplied functions. The programmer need not be an expert in parallel programming to

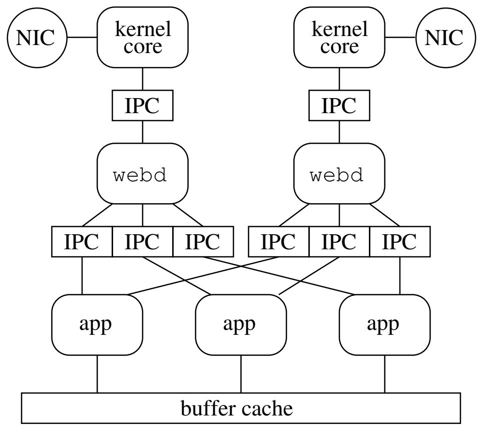


Figure 6: A Corey Web server configuration with two kernel cores, two `webd` cores and three application cores. Rectangles represent segments, rounded rectangles represents pcores, and circles represent devices.

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 started with the Phoenix MapReduce implementation [22], which is optimized for shared-memory multiprocessors. We reimplemented Phoenix to simplify its implementation, to use better algorithms for manipulating the intermediate data, and to optimize its performance. We call this reimplementation Metis.

Metis on Corey exploits address ranges as described in Section 3. Metis uses a separate address space on each core, with private mappings for most data (e.g. local variables and the input to map), so that each core can update its own page tables without contention. Metis uses address ranges to share the output of the map on each core with the reduce phase on other cores. This arrangement avoids contention as each map instance adds pages to hold its intermediate output, and ensures that the reduce phase incurs no soft page faults while processing intermediate data from the map phase.

6.2 Web server applications

The main processing in a Web server includes low-level network device handling, TCP processing, HTTP protocol parsing and formatting, application processing (for dynamically generated content), and access to application data. Much of this processing involves operating system services. Different parts of the processing require different parallelization strategies. For example, HTTP parsing is easy to parallelize by connection, while application processing is often best parallelized by partitioning application data across cores to avoid multiple cores contending for the same data. Even for read-only application data, such partitioning may help maximize the amount of distinct data cached by avoiding duplicating the same data in many cores' local caches.

The Corey Web server is built from three components: Web daemons (`webd`), kernel cores, and applications (see Figure 6). The components communicate via shared-memory IPC. `Webd` is responsible for processing HTTP requests. Every core running a `webd` front-end uses a private TCP/IP stack. `Webd` can manipulate the network device directly or use a kernel core to do so. If using a kernel core, the TCP/IP stack of each core passes packets to transmit and buffers to receive incoming packets to the kernel core using a syscall device. `Webd` parses HTTP requests and hands them off to a core running application code. The application core performs the required computation and returns the results to `webd`. `Webd` packages the results in an HTTP response and transmits it, possibly using a kernel core. Applications may run on dedicated cores (as shown in Figure 6) or run on the same core as a `webd` front-end.

All kernel objects necessary for a `webd` core to complete a request, such as packet buffer segments, network devices, and shared segments used for IPC are referenced by a private per-`webd` core share. Furthermore, most kernel objects, with the exception of the IPC segments, are used by only one core. With this design, once IPC segments have been mapped into the `webd` and application address ranges, cores process requests without contending over any global kernel data or locks, except as needed by the application.

The application running with `webd` can run in two modes: *random* mode and *locality* mode. In random mode, `webd` forwards each request to a randomly chosen application core. In locality mode, `webd` forwards each request to a core chosen from a hash of the name of the data that will be needed in the request. Locality mode increases the probability that the application core will have the needed data in its cache, and decreases redundant caching of the same data.

7 IMPLEMENTATION

Corey runs on AMD Opteron and the Intel Xeon processors. 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 ranges is geared towards architectures with hardware page tables. Address ranges 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 implementation is divided into two parts: the low-level code that implements Corey objects (described in Section 4) and the high-level Unix-like environment (described in Section 5). The low-level code (the kernel) executes in the supervisor protection domain. The high-level Unix-like services are a library that applications link with and execute in application protection domains.

The low-level kernel implementation, which includes architecture specific functions and device drivers, is 11,000 lines of C and 150 lines of assembly. The Unix-like environment, 11,000 lines of C/C++, provides the buffer cache, `cfork`, and TCP/IP stack interface as well as the Corey-specific glue for the uClibc [28] C standard library, lwIP, and the Streamflow [24] dynamic memory allocator. We fixed several bugs in Streamflow and added support for x86-64 processors.

8 Evaluation

This section demonstrates the performance improvements that can be obtained by allowing applications to control sharing. We make these points using several microbenchmarks evaluating address ranges, kernel cores, and shares independently, as well as with the two applications described in Section 6.

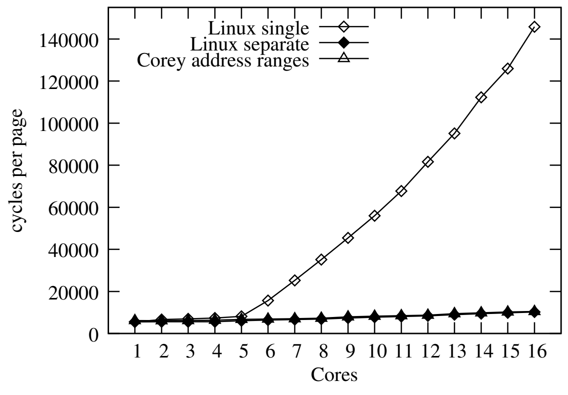
8.1 Experimental setup

We ran all experiments on an AMD 16-core system (see Figure 2 in Section 2) with 64 Gbytes of memory. We counted the number of cache misses and computed the average latency of cache misses using the AMD hardware event counters. All Linux experiments use Debian Linux with kernel version 2.6.25 and pin one thread to each core. The kernel is patched with perfctr 2.6.35 to allow application access to hardware event counters. For Linux MapReduce experiments we used our version of Streamflow because it provided better performance than other allocators we tried, such as TCMalloc [11] and glibc 2.7 malloc. All network experiments were performed on a gigabit switched network, using the server's Intel Pro/1000 Ethernet device.

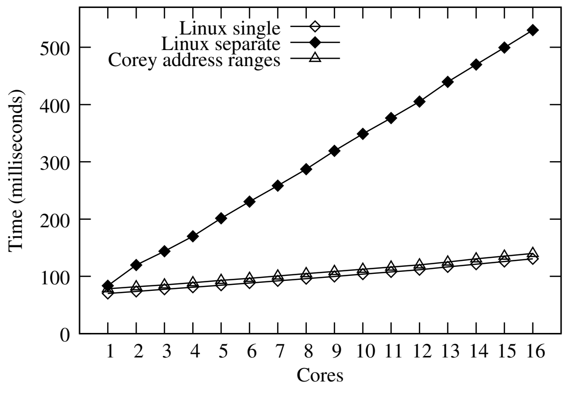
We also have run many of the experiments with Corey and Linux on a 16-core Intel machine and with Windows on 16-core AMD and Intel machines, and we draw conclusions similar to the ones reported in this section.

8.2 Address ranges

To evaluate the benefits of address ranges in Corey, we need to investigate two costs in multicore applications where some memory is private and some is shared. First, the contention costs of manipulating mappings for private memory. Second, the soft page-fault costs for memory that is used on multiple cores. We expect Corey to have low costs for both situations. We expect other systems (represented by Linux in these experiments) to have low cost for only one type of sharing, but not both, depending on whether the application uses a single address space shared by all cores or a separate address space per core.



(a) `memclone`



(b) `mempass`

Figure 7: Address ranges microbenchmark results.

The `memclone` [3] benchmark explores the costs of private memory. `Memclone` has each core allocate a 100 Mbyte array and modify each page of its array. The memory is demand-zero-fill: the kernel initially allocates no memory, and allocates pages only in response to page faults. The kernel allocates the memory from the DRAM system connected to the core's chip. The benchmark measures the average time to modify each page. `Memclone` allocates cores from chips in a round-robin fashion. For example, when using five cores, `memclone` allocates two cores from one chip and one core from each of the other three chips.

Figure 7(a) presents the results for three situations: Linux with a single address space shared by per-core threads, Linux with a separate address space (process) per core but with the 100 Mbyte arrays `mmap`ed into each process, and Corey with separate address spaces but with the arrays mapped via shared address ranges.

`Memclone` scales well on Corey and on Linux with separate address spaces, but poorly on Linux with a single address space. On a page fault both Corey and Linux verify that the faulting address is valid, allocate and clear a 4 Kbyte physical page, and insert the physical page into the hardware page table. Clearing a 4 Kbyte page incurs 64 L3 cache misses and copies 4 Kbytes from DRAM to the local L1 cache and writes 4 Kbytes from the L3 back to DRAM; however, the AMD cache line prefetcher allows a core to clear a page in 3350 cycles (or at a rate of 1.2 bytes per cycle).

Corey and Linux with separate address spaces each incur only 64 cache misses per page fault, regardless of the number of cores. However, the cycles per page gradually increases because the per-chip memory controller can only clear 1.7 bytes per cycle and is saturated when `memclone` uses more than two cores on a chip.

For Linux with a single address space cores contend over shared address space data structures. With six cores the cache-coherency messages generated from locking and manipulating the shared address space begin to congest the interconnect, which increases the cycles required for processing a page fault. Processing a page fault takes 14 times longer with 16 cores than with one.

We use a benchmark called `mempass` to evaluate the costs of memory that is used on multiple cores. `Mempass` allocates a single 100 Mbyte shared buffer on one of the cores, touches each page of the buffer, and then passes it to another core, which repeats the process until every core touches every page. `Mempass` measures the total time for every core to touch every page.

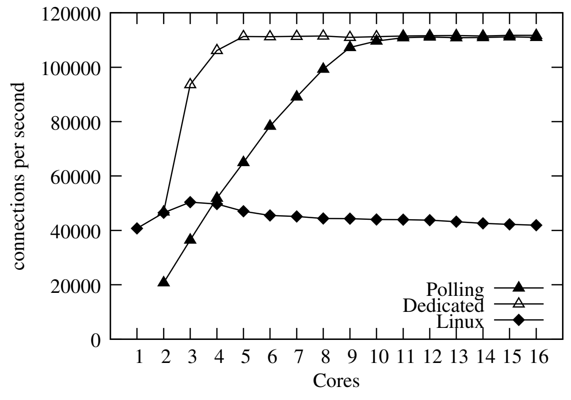
Figure 7(b) presents the results for the same configurations used in `memclone`. This time Linux with a single address space performs well, while Linux with separate address spaces performs poorly. Corey performs well here too. Separate address spaces are costly with this workload because each core separately takes a soft page fault for the shared page.

To summarize, a Corey application can use address ranges to get good performance for both shared and private memory. In contrast, a Linux application can get good performance for only one of these types of memory.

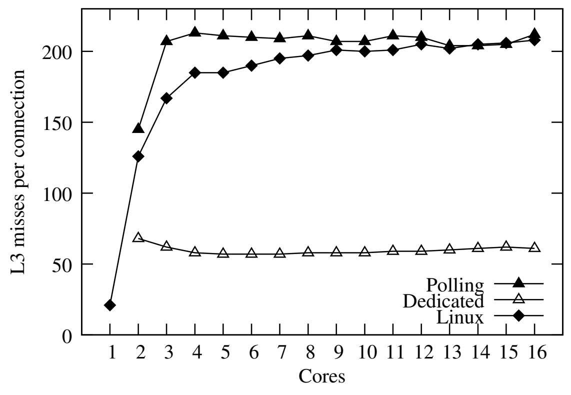
8.3 Kernel cores

This section explores an example in which use of the Corey kernel core abstraction improves scalability. The benchmark application is a simple TCP service, which accepts incoming connections, writing 128 bytes to each connection before closing it. Up to 15 separate client machines (as many as there are active server cores) generate the connection requests.

We compare two server configurations. One of them (called "Dedicated") uses a kernel core to handle all network device processing: placing packet buffer pointers in device DMA descriptors, polling for received packets and transmit completions, triggering device transmissions, and manipulating corresponding driver data structures. The second configuration, called "Polling", uses a kernel core only to poll for received packet notifications and transmit completions. In both cases, each other core runs a private TCP/IP stack and an instance of the TCP service. For Dedicated, each service core uses shared-memory IPC to send and receive packet buffers with the kernel core. For Polling, each service core transmits packets and registers receive packet buffers by directly manipulating the device DMA descriptors (with locking), and is notified of received packets via IPC from the Polling kernel core. The purpose of the comparison is to show the effect on performance of moving all device processing to the Dedicated kernel core, thereby eliminating contention over device driver data structures. Both configurations poll for received packets, since otherwise interrupt costs would dominate performance.



(a) Throughput.



(b) L3 cache misses.

Figure 8: TCP microbenchmark results.

Figure 8(a) presents the results of the TCP benchmark. The network device appears to be capable of handling only about 900,000 packets per second in total, which limits the throughput to about 110,000 connections per second in all configurations (each connection involves 4 input and 4 output packets). The dedicated configuration reaches 110,000 with only five cores, while Polling requires 11 cores. That is, each core is able to handle more connections per second in the Dedicated configuration than in Polling.

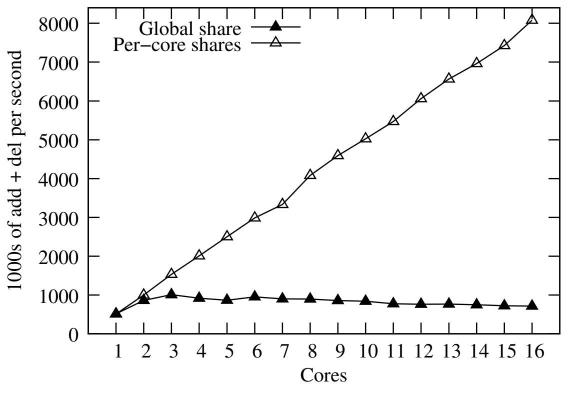
Most of this difference in performance is due to the Dedicated configuration incurring fewer L3 misses. Figure 8(b) shows the number of L3 misses per connection. Dedicated incurs about 50 L3 misses per connection, or slightly more than six per packet. These include misses to manipulate the IPC data structures (shared with the core that sent or received the packet) and the device DMA descriptors (shared with the device) and the misses for a core to read a received packet. Polling incurs about 25 L3 misses per packet; the difference is due to misses on driver data structures (such as indexes into the DMA descriptor rings) and the locks that protect them. To process a packet the device driver must acquire and release an MCS lock twice: once to place the packet on DMA ring buffer and once to remove it. Since each L3 miss takes about 100 nanoseconds to service, misses account for slightly less than half the total cost for the Polling configuration to process a connection (with two cores). This analysis is consistent with Dedicated processing about twice as many connections with two cores, since Dedicated has many fewer misses. This approximate relationship holds up to the 110,000 connection per second limit.

The Linux results are presented for reference. In Linux every core shares the same network stack and a core polls the network device when the packet rate is high. All cores place output packets on a software transmit queue. Packets may be removed from the software queue by any core, but usually the polling core transfers the packets from the software queue to the hardware transmit buffer on the network device. On packet reception, the polling core removes the packet from the ring buffer and performs the processing necessary to place the packet on the queue of a TCP socket where it can be read by the application. With two cores, most cycles on both cores are used by the kernel; however, as the number of cores increases the polling core quickly becomes a bottleneck and other cores have many idle cycles.

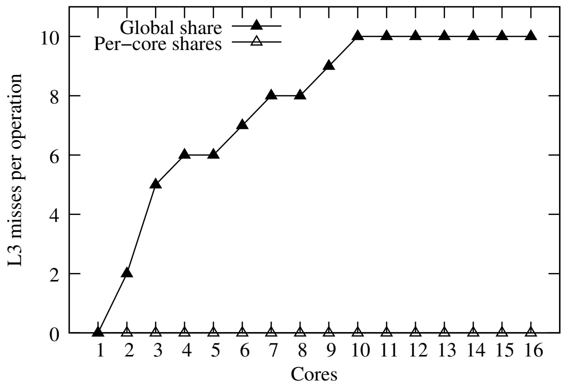
The TCP microbenchmark results demonstrate that a Corey application can use kernel cores to improve scalability by dedicating cores to handling kernel functions and data.

8.4 Shares

We demonstrate the benefits of shares on Corey by comparing the results of two microbenchmarks. In the first microbenchmark each core calls `share_addobj` to add a per-core segment to a global share and then removes the segment from the share with `share_delobj`. The benchmark measures the throughput of add and remove operations. As the number of cores increases, cores contend on the scalable read-write lock protecting the hashtable used to implement the share. The second microbenchmark is similar to the first, except each core adds a per-core segment to a local share so that the cores do not contend.



(a) Throughput.



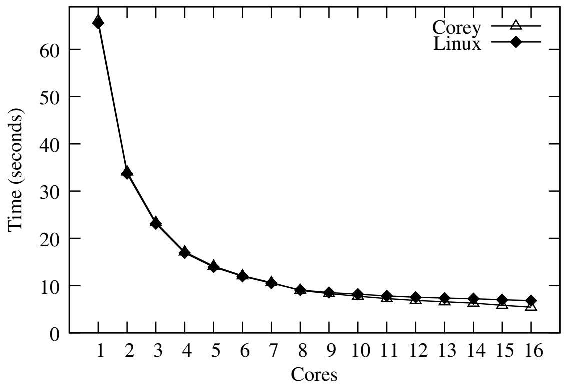
(b) L3 cache misses.

Figure 9: Share microbenchmark results.

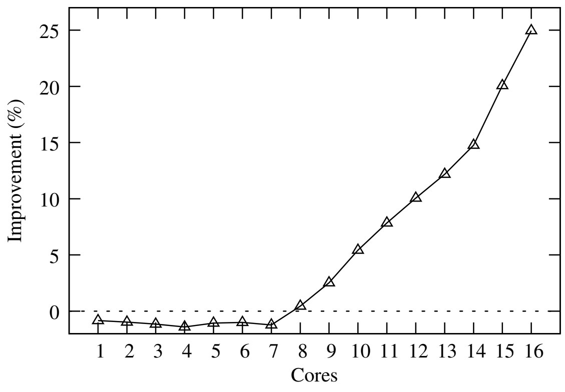
Figure 9 presents the results. The global share scales relatively well for the first three cores, but at four cores performance begins to slowly decline. Cores contend on the global share, which increases the amount of cache coherence traffic and makes manipulating the global share more expensive. The local shares scale perfectly. At 16 cores adding to and removing from the global share costs 10 L3 cache misses, while no L3 cache misses result from the pair of operations in a local share.

We measured the L3 misses for the Linux file descriptor microbenchmark described in Section 1. On one core a duplicate and close incurs no L3 cache misses; however, L3 misses increase with the number of cores and at 16 cores a `dup` and `close` costs 51 L3 cache misses.

A Corey application can use shares to avoid bottlenecks that might be caused by contention on kernel data structures when creating and manipulating kernel objects. A Linux application, on the other hand, must use sharing policies specified by kernel implementers, which



(a) Corey and Linux performance.



(b) Corey improvement over Linux.

Figure 10: MapReduce `wri` results.

might force inter-core sharing and unnecessarily limit scalability.

8.5 Applications

To demonstrate that address ranges, kernel cores, and shares have an impact on application performance we presents benchmark results from the Corey port of Metis and from `webd`.

8.5.1 MapReduce

We compare Metis on Corey and on Linux, using the `wri` MapReduce application to build a reverse index of the words in a 1 Gbyte file. Metis allocates 2 Gbytes to hold intermediate values and like `memclone`, Metis allocates cores from chips in a round-robin fashion.

On Linux, cores share a single address space. On Corey each core maps the memory segments holding intermediate results using per-core shared address ranges. During the map phase each core writes $\langle \text{word}, \text{index} \rangle$ key-value pairs to its per-core intermediate results memory. For pairs with the same word, the indices are grouped as an array. More than 80% of the map phase is spent in `strcmp`, leaving little scope for improvement from address ranges. During the reduce phase each core copies all the indices for each word it is responsible

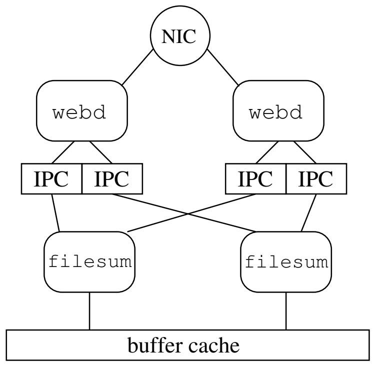


Figure 11: A `webd` configuration with two front-end cores and two `filesum` cores. Rectangles represent segments, rounded rectangles represents pcores, and the circle represents a network device.

for from the intermediate result memories of all cores to a freshly allocated buffer. The reduce phase spends most of its time copying memory and handling soft page faults. As the number of cores increases, Linux cores contend while manipulating address space data while Corey's address ranges keep contention costs low.

Figure 10(a) presents the absolute performance of `wri` on Corey and Linux and Figure 10(b) shows Corey's improvement relative to Linux. For less than eight cores Linux is 1% faster than Corey because Linux's soft page fault handler is about 10% faster than Corey's when there is no contention. With eight cores on Linux, address space contention costs during reduce cause execution time for reduce to increase from 0.47 to 0.58 seconds, while Corey's reduce time decreases from 0.46 to 0.42 seconds. The time to complete the reduce phase continues to increase on Linux and decrease on Corey as more cores are used. At 16 cores the reduce phase takes 1.9 seconds on Linux and only 0.35 seconds on Corey.

8.5.2 `Webd`

As described in Section 6.2, applications might be able to increase performance by dedicating application data to cores. We explore this possibility using a `webd` application called `filesum`. `Filesum` expects a file name as argument, and returns the sum of the bytes in that file. `Filesum` can be configured in two modes. "Random" mode sums the file on a random core and "Locality" mode assigns each file to a particular core and sums a requested file on its assigned core. Figure 11 presents a four core configuration.

We measured the throughput of `webd` with `filesum` in Random mode and in Locality mode. We configured `webd` front-end servers on eight cores and `filesum` on eight cores. This experiment did not use a kernel core or device polling. A single core handles network interrupts, and all cores running `webd` directly manipulate the network device. Clients only request files from a set of eight, so each `filesum` core is assigned one file, which

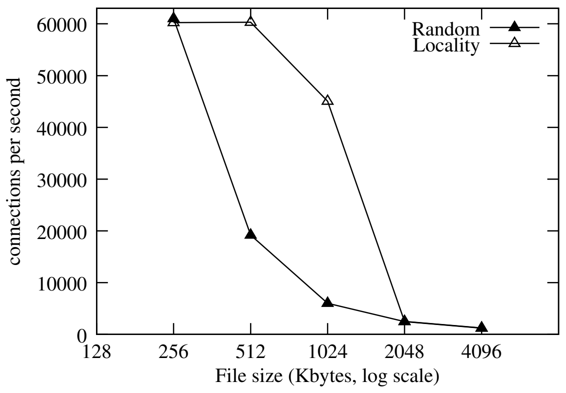


Figure 12: Throughput of Random mode and Locality mode.

it maps into memory. The HTTP front-end cores and `filesum` cores exchange file requests and responses using a shared memory segment.

Figure 12 presents the results of Random mode and Locality mode. For file sizes of 256 Kbytes and smaller the performance of Locality and Random mode are both limited by the performance of the `webd` front-ends' network stacks. In Locality mode, each `filesum` core can store its assigned 512 Kbyte or 1024 Kbyte file in chip caches. For 512 Kbyte files Locality mode achieves 3.1 times greater throughput than Random and for 1024 Kbyte files locality mode achieves 7.5 times greater throughput. Larger files cannot be fit in chip caches and the performance of both Locality mode and Random mode is limited by DRAM performance.

9 Discussion and future work

The results above should be viewed as a case for the principle that applications should control sharing rather than a conclusive "proof". Corey lacks many features of commodity operating systems, such as Linux, which influences experimental results both positively and negatively.

Many of the ideas in Corey could be applied to existing operating systems such as Linux. The Linux kernel could use a variant of address ranges internally, and perhaps allow application control via `mmap` flags. Applications could use kernel-core-like mechanisms to reduce system call invocation costs, to avoid concurrent access to kernel data, or to manage an application's sharing of its own data using techniques like computation migration [6, 14]. Finally, it may be possible for Linux to provide share-like control over the sharing of file descriptors, entries in buffer and directory lookup caches, and network stacks.

10 Related work

Corey is influenced by Disco [5] (and its relatives [13, 30]), which runs as a virtual machine monitor on a NUMA machine. Like Disco, Corey aims to avoid kernel bottlenecks with a small kernel that minimizes shared data. However, Corey has a kernel interface like an exokernel [9] rather than a virtual machine monitor.

Corey's focus on supporting library operating systems resembles Disco's SPLASHOS, a runtime tailored for running the Splash benchmark [26]. Corey's library operating systems require more operating system features, which has led to the Corey kernel providing a number of novel ideas to allow libraries to control sharing (address ranges, kernel cores, and shares).

K42 [3] and Tornado [10], like Corey, are designed so that independent applications do not contend over resources managed by the kernel. The K42 and Tornado kernels provide clustered objects that allow different implementations based on sharing patterns. The Corey kernel takes a different approach; applications customize sharing policies as needed instead of selecting from a set of policies provided by kernel implementers. 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 multiprocessors [4]. The Linux community has made many changes to improve scalability (e.g., Read-Copy-Update (RCU) [20] locks, local runqueues [1], `libnuma` [16], improved load-balancing support [17]). These techniques are complementary to the new techniques that Corey introduces and 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.

As mentioned in the Introduction, Gough *et al.* [12] and Veal and Foong [29] have identified Linux scaling challenges in the kernel implementations of scheduling and directory lookup, respectively.

Saha *et al.* have proposed the Multi-Core Run Time (McRT) for desktop workloads [23] and have explored dedicating some cores to McRT and the application and others to the operating system. 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.

An Intel white paper reports use of spatial multiplexing of packet processing in a network analyzer [15], with performance considerations similar to `webd` running in Locality or Random modes.

Effective use of multicore processors has many aspects and potential solution approaches; Corey explores only a few. For example, applications could be assisted in coping with heterogeneous hardware [25]; threads could be automatically assigned to cores in a way that minimizes cache misses [27]; or applications could schedule work in order to minimize the number of times a given datum needs to be loaded from DRAM [7].

11 CONCLUSIONS

This paper argues that, in order for applications to scale on multicore architectures, applications must control sharing. Corey is a new kernel that follows this principle. Its address range, kernel core, and share abstractions ensure that each kernel data structure is used by only one core by default, while giving applications the ability to specify when sharing of kernel data is necessary. Experiments with a MapReduce application and a synthetic Web application demonstrate that Corey's design allows these applications to avoid scalability bottlenecks in the operating system and outperform Linux on 16-core machines. We hope that the Corey ideas will help applications to scale to the larger number of cores on future processors. All Corey source code is publicly available.

ACKNOWLEDGMENTS

We thank Russ Cox, Austin Clements, Evan Jones, Emil Sit, Xi Wang, and our shepherd, Wolfgang Schröder-Preikschat, for their feedback. The NSF partially funded this work through award number 0834415.

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] 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.

[5] 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.

[6] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[7] 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.

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

[9] 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.

[10] 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.

[11] Google Performance Tools. `http://goog-perftools.sourceforge.net/`.

[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] W. C. Hsieh, M. F. Kaashoek, and W. E. Weihl. Dynamic computation migration in dsm systems. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 1996.

[15] Intel. Supra-linear Packet Processing Performance with Intel Multi-core Processors. `ftp://download.intel.com/technology/advanced_comm/31156601.pdf`.

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

[17] Linux kernel mailing list. `http://kerneltrap.org/node/8059`.

[18] Linux Symposium. `http://www.linuxsymposium.org/`.

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

[20] 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.

[21] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[22] 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.

[23] 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.

[24] 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.

[25] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)*, Boston, MA, USA, June 2008.

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

[27] 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.

[28] uClibc. `http://www.uclibc.org/`.

[29] 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.

[30] 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.

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