An Analysis of Linux Scalability to Many Cores

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev,
M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich
MIT CSAIL

ABSTRACT

This paper analyzes the scalability of seven system applications (Exim, memcached, Apache, PostgreSQL, gmake, Psearchy, and MapReduce) running on Linux on a 48-core computer. Except for gmake, all applications trigger scalability bottlenecks inside a recent Linux kernel. Using mostly standard parallel programming techniques—this paper introduces one new technique, *sloppy counters*—these bottlenecks can be removed from the kernel or avoided by changing the applications slightly. Modifying the kernel required in total 3002 lines of code changes. A speculative conclusion from this analysis is that there is no scalability reason to give up on traditional operating system organizations just yet.

1 INTRODUCTION

There is a sense in the community that traditional kernel designs won't scale well on multicore processors: that applications will spend an increasing fraction of their time in the kernel as the number of cores increases. Prominent researchers have advocated rethinking operating systems [10, 28, 43] and new kernel designs intended to allow scalability have been proposed (e.g., Barrelfish [11], Corey [15], and fos [53]). This paper asks whether traditional kernel designs can be used and implemented in a way that allows applications to scale.

This question is difficult to answer conclusively, but we attempt to shed a small amount of light on it. We analyze scaling a number of system applications on Linux running with a 48-core machine. We examine Linux because it has a traditional kernel design, and because the Linux community has made great progress in making it scalable. The applications include the Exim mail server [2], memcached [3], Apache serving static files [1], PostgreSQL [4], gmake [23], the Psearchy file indexer [35, 48], and a multicore MapReduce library [38]. These applications, which we will refer to collectively as MOSBENCH, are designed for parallel execution and stress many major Linux kernel components.

Our method for deciding whether the Linux kernel design is compatible with application scalability is as follows. First we measure scalability of the MOSBENCH applications on a recent Linux kernel (2.6.35-rc5, released July 12, 2010) with 48 cores, using the in-memory `tmpfs` file system to avoid disk bottlenecks. gmake scales well, but the other applications scale poorly, performing much less work per core with 48 cores than with one core. We attempt to understand and fix the scalability problems, by modifying either the applications or the Linux kernel. We then iterate, since fixing one scalability problem usually exposes further ones. The end result for each application is either good scalability on 48 cores, or attribution of non-scalability to a hard-to-fix problem with the application, the Linux kernel, or the underlying hardware. The analysis of whether the kernel design is compatible with scaling rests on the extent to which our changes to the Linux kernel turn out to be modest, and the extent to which hard-to-fix problems with the Linux kernel ultimately limit application scalability.

As part of the analysis, we fixed three broad kinds of scalability problems for MOSBENCH applications: problems caused by the Linux kernel implementation, problems caused by the applications' user-level design, and problems caused by the way the applications use Linux kernel services. Once we identified a bottleneck, it typically required little work to remove or avoid it. In some cases we modified the application to be more parallel, or to use kernel services in a more scalable fashion, and in others we modified the kernel. The kernel changes are all localized, and typically involve avoiding locks and atomic instructions by organizing data structures in a distributed fashion to avoid unnecessary sharing. One reason the required changes are modest is that stock Linux already incorporates many modifications to improve scalability. More speculatively, perhaps it is the case that Linux's system-call API is well suited to an implementation that avoids unnecessary contention over kernel objects.

The main contributions of this paper are as follows. The first contribution is a set of 16 scalability improvements to the Linux 2.6.35-rc5 kernel, resulting in what we refer to as the patched kernel, PK. A few of the changes rely on a new idea, which we call *sloppy counters*, that has the nice property that it can be used to augment shared counters to make some uses more scalable without having to change all uses of the shared counter. This technique is particularly effective in Linux because typically only a few uses of a given shared counter are scalability bottlenecks; sloppy counters allow us to replace just those few uses without modifying the many other uses in the kernel. The second contribution is a set of application

benchmarks, MOSBENCH, to measure scalability of operating systems, which we make publicly available. The third is a description of the techniques required to improve the scalability of the MOSBENCH applications. Our final contribution is an analysis using MOSBENCH that suggests that there is no immediate scalability reason to give up on traditional kernel designs.

The rest of the paper is organized as follows. Section 2 relates this paper to previous work. Section 3 describes the applications in MOSBENCH and what operating system components they stress. Section 4 summarizes the differences between the stock and PK kernels. Section 5 reports on the scalability of MOSBENCH on the stock Linux 2.6.35-rc5 kernel and the PK kernel. Section 6 discusses the implications of the results. Section 7 summarizes this paper's conclusions.

2 RELATED WORK

There is a long history of work in academia and industry to scale Unix-like operating systems on shared-memory multiprocessors. Research projects such as the Stanford FLASH [33] as well as companies such as IBM, Sequent, SGI, and Sun have produced shared-memory machines with tens to hundreds processors running variants of Unix. Many techniques have been invented to scale software for these machines, including scalable locking (e.g., [41]), wait-free synchronization (e.g., [27]), multiprocessor schedulers (e.g., [8, 13, 30, 50]), memory management (e.g., [14, 19, 34, 52, 57]), and fast message passing using shared memory (e.g., [12, 47]). Textbooks have been written about adapting Unix for multiprocessors (e.g., [46]). These techniques have been incorporated in current operating systems such as Linux, Mac OS X, Solaris, and Windows. Cantrill and Bonwick summarize the historical context and real-world experience [17].

This paper extends previous scalability studies by examining a large set of systems applications, by using a 48-core PC platform, and by detailing a particular set of problems and solutions in the context of Linux. These solutions follow the standard parallel programming technique of factoring data structures so that each core can operate on separate data when sharing is not required, but such that cores can share data when necessary.

Linux scalability improvements. Early multiprocessor Linux kernels scaled poorly with kernel-intensive parallel workloads because the kernel used coarse-granularity locks for simplicity. Since then the Linux community has redesigned many kernel subsystems to improve scalability (e.g., Read-Copy-Update (RCU) [39], local run queues [6], `libnuma` [31], and improved load-balancing support [37]). The Linux symposium (`www.linuxsymposium.org`) features papers related to scalability almost every year. Some of the redesigns are based on the above-mentioned research, and some companies, such as IBM and SGI [16], have contributed code directly. Kleen provides a brief history of Linux kernel modifications for scaling and reports some areas of poor scalability in a recent Linux version (2.6.31) [32]. In this paper, we identify additional kernel scaling problems and describes how to address them.

Linux scalability studies. Gough *et al.* study the scalability of Oracle Database 10g running on Linux 2.6.18 on dual-core Intel Itanium processors [24]. The study finds problems with the Linux run queue, slab allocator, and I/O processing. Cui *et al.* uses the TPCC-UVa and Sysbench-OLTP benchmarks with PostgreSQL to study the scalability of Linux 2.6.25 on an Intel 8-core system [56], and finds application-internal bottlenecks as well as poor kernel scalability in System V IPC. We find that these problems have either been recently fixed by the Linux community or are a consequence of fixable problems in PostgreSQL.

Veal and Foong evaluate the scalability of Apache running on Linux 2.6.20.3 on an 8-core AMD Opteron computer using SPECweb2005 [51]. They identify Linux scaling problems in the kernel implementations of scheduling and directory lookup, respectively. On a 48-core computer, we also observe directory lookup as a scalability problem and PK applies a number of techniques to address this bottleneck. Pesterev *et al.* identify scalability problems in the Linux 2.6.30 network code using memcached and Apache [44]. The PK kernel addresses these problems by using a modern network card that supports a large number of virtual queues (similar to the approach taken by Route Bricks [21]).

Cui *et al.* describe microbenchmarks for measuring multicore scalability and report results from running them on Linux on a 32-core machine [55]. They find a number of scalability problems in Linux (e.g., memory-mapped file creation and deletion). Memory-mapped files show up as a scalability problem in one MOSBENCH application when multiple threads run in the same address space with memory-mapped files.

A number of new research operating systems use scalability problems in Linux as motivation. The Corey paper [15] identified bottlenecks in the Linux file descriptor and virtual memory management code caused by unnecessary sharing. Both of these bottlenecks are also triggered by MOSBENCH applications. The Barrelfish paper [11] observed that Linux TLB shootdown scales poorly. This problem is not observed in the MOSBENCH applications. Using microbenchmarks, the fos paper [53] finds that the physical page allocator in Linux 2.6.24.7 does not scale beyond 8 cores and that executing the kernel and applications on the same core results in cache interference and high miss rates. We find that the page allocator isn't a bottleneck for MOSBENCH applications on 48 cores (even though they stress memory allocation), though we have

reason to believe it would be a problem with more cores. However, the problem appears to be avoidable by, for example, using super-pages or modifying the kernel to batch page allocation.

Solaris scalability studies. Solaris provides a UNIX API and runs on SPARC-based and x86-based multicore processors. Solaris incorporates SNZIs [22], which are similar to sloppy counters (see section 4.3). Tseng *et al.* report that SAP-SD, IBM Trade and several synthetic benchmarks scale well on an 8-core SPARC system running Solaris 10 [49]. Zou *et al.* encountered coarse grained locks in the UDP networking stack of Solaris 10 that limited scalability of the OpenSER SIP proxy server on an 8-core SPARC system [29]. Using the microbenchmarks mentioned above [55], Cui *et al.* compare FreeBSD, Linux, and Solaris [54], and find that Linux scales better on some microbenchmarks and Solaris scales better on others. We ran some of the MOSBENCH applications on Solaris 10 on the 48-core machine used for this paper. While the Solaris license prohibits us from reporting quantitative results, we observed similar or worse scaling behavior compared to Linux; however, we don't know the causes or whether Solaris would perform better on SPARC hardware. We hope, however, that this paper helps others who might analyze Solaris.

3 THE MOSBENCH APPLICATIONS

To stress the kernel we chose two sets of applications: 1) applications that previous work has shown not to scale well on Linux (memcached; Apache; and Metis, a MapReduce library); and 2) applications that are designed for parallel execution and are kernel intensive (gmake, PostgreSQL, Exim, and Psearchy). Because many applications are bottlenecked by disk writes, we used an in-memory `tmpfs` file system to explore non-disk limitations. We drive some of the applications with synthetic user workloads designed to cause them to use the kernel intensively, with realism a secondary consideration. This collection of applications stresses important parts of many kernel components (e.g., the network stack, file name cache, page cache, memory manager, process manager, and scheduler). Most spend a significant fraction of their CPU time in the kernel when run on a single core. All but one encountered serious scaling problems at 48 cores caused by the stock Linux kernel. The rest of this section describes the selected applications, how they are parallelized, and what kernel services they stress.

3.1 Mail server

Exim [2] is a mail server. We operate it in a mode where a single master process listens for incoming SMTP connections via TCP and forks a new process for each connection, which in turn accepts the incoming mail, queues it in a shared set of spool directories, appends it to the per-user mail file, deletes the spooled mail, and records the delivery in a shared log file. Each per-connection process also forks twice to deliver each message. With many concurrent client connections, Exim has a good deal of parallelism. It spends 69% of its time in the kernel on a single core, stressing process creation and small file creation and deletion.

3.2 Object cache

memcached [3] is an in-memory key-value store often used to improve web application performance. A single memcached server running on multiple cores is bottlenecked by an internal lock that protects the key-value hash table. To avoid this problem, we run multiple memcached servers, each on its own port, and have clients deterministically distribute key lookups among the servers. This organization allows the servers to process requests in parallel. When request sizes are small, memcached mainly stresses the network stack, spending 80% of its time processing packets in the kernel at one core.

3.3 Web server

Apache [1] is a popular Web server, which previous work (e.g., [51]) has used to study Linux scalability. We run a single instance of Apache listening on port 80. We configure this instance to run one process per core. Each process has a thread pool to service connections; one thread is dedicated to accepting incoming connections while the other threads process the connections. In addition to the network stack, this configuration stresses the file system (in particular directory name lookup) because it stats and opens a file on every request. Running on a single core, an Apache process spends 60% of its execution time in the kernel.

3.4 Database

PostgreSQL [4] is a popular open source SQL database, which, unlike many of our other workloads, makes extensive internal use of shared data structures and synchronization. PostgreSQL also stresses many shared resources in the kernel: it stores database tables as regular files accessed concurrently by all PostgreSQL processes, it starts one process per connection, it makes use of kernel locking interfaces to synchronize and load balance these processes, and it communicates with clients over TCP sockets that share the network interface.

Ideally, PostgreSQL would scale well for read-mostly workloads, despite its inherent synchronization needs. PostgreSQL relies on snapshot isolation, a form of optimistic concurrency control that avoids most read locks. Furthermore, most write operations acquire only row-level locks exclusively and acquire all coarser-grained locks in shared modes. Thus, in principle, PostgreSQL should exhibit little contention for read-mostly workloads. In practice, PostgreSQL is limited by bottlenecks in both

its own code and in the kernel. For a read-only workload that avoids most application bottlenecks, PostgreSQL spends only 1.5% of its time in the kernel with one core, but this grows to 82% with 48 cores.

3.5 Parallel build

gmake [23] is an implementation of the standard make utility that supports executing independent build rules concurrently. gmake is the unofficial default benchmark in the Linux community since all developers use it to build the Linux kernel. Indeed, many Linux patches include comments like "This speeds up compiling the kernel." We benchmarked gmake by building the stock Linux 2.6.35-rc5 kernel with the default configuration for x86_64. gmake creates more processes than there are cores, and reads and writes many files. The execution time of gmake is dominated by the compiler it runs, but system time is not negligible: with one core, 7.6% of the execution time is system time.

3.6 File indexer

Psearchy is a parallel version of searchy [35, 48], a program to index and query Web pages. We focus on the indexing component of searchy because it is more system intensive. Our parallel version, pedsort, runs the searchy indexer on each core, sharing a work queue of input files. Each core operates in two phases. In phase 1, it pulls input files off the work queue, reading each file and recording the positions of each word in a per-core hash table. When the hash table reaches a fixed size limit, it sorts it alphabetically, flushes it to an intermediate index on disk, and continues processing input files. Phase 1 is both compute intensive (looking up words in the hash table and sorting it) and file-system intensive (reading input files and flushing the hash table). To avoid stragglers in phase 1, the initial work queue is sorted so large files are processed first. Once the work queue is empty, each core merges the intermediate index files it produced, concatenating the position lists of words that appear in multiple intermediate indexes, and generates a binary file that records the positions of each word and a sequence of Berkeley DB files that map each word to its byte offset in the binary file. To simplify the scalability analysis, each core starts a new Berkeley DB every 200,000 entries, eliminating a logarithmic factor and making the aggregate work performed by the indexer constant regardless of the number of cores. Unlike phase 1, phase 2 is mostly file-system intensive. While pedsort spends only 1.9% of its time in the kernel at one core, this grows to 23% at 48 cores, indicating scalability limitations.

3.7 MapReduce

Metis is a MapReduce [20] library for single multicore servers inspired by Phoenix [45]. We use Metis with an application that generates inverted indices. This workload allocates large amounts of memory to hold temporary tables, stressing the kernel memory allocator and soft page fault code. This workload spends 3% of its runtime in the kernel with one core, but this rises to 16% at 48 cores.

4 KERNEL OPTIMIZATIONS

The MOSBENCH applications trigger a few scalability bottlenecks in the kernel. We describe the bottlenecks and our solutions here, before presenting detailed per-application scaling results in Section 5, because many of the bottlenecks are common to multiple applications. Figure 1 summarizes the bottlenecks. Some of these problems have been discussed on the Linux kernel mailing list and solutions proposed; perhaps the reason these solutions have not been implemented in the standard kernel is that the problems are not acute on small-scale SMPs or are masked by I/O delays in many applications. Figure 1 also summarizes our solution for each bottleneck.

4.1 Scalability tutorial

Why might one expect performance to scale well with the number of cores? If a workload consists of an unlimited supply of tasks that do not interact, then you'd expect to get linear increases in total throughput by adding cores and running tasks in parallel. In real life parallel tasks usually interact, and interaction usually forces serial execution. Amdahl's Law summarizes the result: however small the serial portion, it will eventually prevent added cores from increasing performance. For example, if 25% of a program is serial (perhaps inside some global locks), then any number of cores can provide no more than 4-times speedup.

Here are a few types of serializing interactions that the MOSBENCH applications encountered. These are all classic considerations in parallel programming, and are discussed in previous work such as [17].

- The tasks may lock a shared data structure, so that increasing the number of cores increases the lock wait time.
- The tasks may write a shared memory location, so that increasing the number of cores increases the time spent waiting for the cache coherence protocol to fetch the cache line in exclusive mode. This problem can occur even in lock-free shared data structures.
- The tasks may compete for space in a limited-size shared hardware cache, so that increasing the number of cores increases the cache miss rate. This problem can occur even if tasks never share memory.
- The tasks may compete for other shared hardware resources such as inter-core interconnect or DRAM

Parallel `accept`			Apache
Concurrent `accept` system calls contend on shared `socket` fields.	⇒	User per-core backlog queues for listening sockets.	
`dentry` reference counting			Apache, Exim
File name resolution contends on directory entry reference counts.	⇒	Use sloppy counters to reference count directory entry objects.	
Mount point (`vfsmount`) reference counting			Apache, Exim
Walking file name paths contends on mount point reference counts.	⇒	Use sloppy counters for mount point objects.	
IP packet destination (`dst_entry`) reference counting			memcached, Apache
IP packet transmission contends on routing table entries.	⇒	Use sloppy counters for IP routing table entries.	
Protocol memory usage tracking			memcached, Apache
Cores contend on counters for tracking protocol memory consumption.	⇒	Use sloppy counters for protocol usage counting.	
Acquiring directory entry (`dentry`) spin locks			Apache, Exim
Walking file name paths contends on per-directory entry spin locks.	⇒	Use a lock-free protocol in `dlookup` for checking filename matches.	
Mount point table spin lock			Apache, Exim
Resolving path names to mount points contends on a global spin lock.	⇒	Use per-core mount table caches.	
Adding files to the open list			Apache, Exim
Cores contend on a per-super block list that tracks open files.	⇒	Use per-core open file lists for each super block that has open files.	
Allocating DMA buffers			memcached, Apache
DMA memory allocations contend on the memory node 0 spin lock.	⇒	Allocate Ethernet device DMA buffers from the local memory node.	
False sharing in `net_device` and `device`			memcached, Apache, PostgreSQL
False sharing causes contention for read-only structure fields.	⇒	Place read-only fields on their own cache lines.	
False sharing in `page`			Exim
False sharing causes contention for read-mostly structure fields.	⇒	Place read-only fields on their own cache lines.	
`inode` lists			memcached, Apache
Cores contend on global locks protecting lists used to track `inode`s.	⇒	Avoid acquiring the locks when not necessary.	
Dcache lists			memcached, Apache
Cores contend on global locks protecting lists used to track `dentry`s.	⇒	Avoid acquiring the locks when not necessary.	
Per-`inode` mutex			PostgreSQL
Cores contend on a per-`inode` mutex in `lseek`.	⇒	Use atomic reads to eliminate the need to acquire the mutex.	
Super-page fine grained locking			Metis
Super-page soft page faults contend on a per-process mutex.	⇒	Protect each super-page memory mapping with its own mutex.	
Zeroing super-pages			Metis
Zeroing super-pages flushes the contents of on-chip caches.	⇒	Use non-caching instructions to zero the contents of super-pages.	

Figure 1: A summary of Linux scalability problems encountered by MOSBENCH applications and their corresponding fixes. The fixes add 2617 lines of code to Linux and remove 385 lines of code from Linux.

interfaces, so that additional cores spend their time waiting for those resources rather than computing.

- There may be too few tasks to keep all cores busy, so that increasing the number of cores leads to more idle cores.

Many scaling problems manifest themselves as delays caused by cache misses when a core uses data that other cores have written. This is the usual symptom both for lock contention and for contention on lock-free mutable data. The details depend on the hardware cache coherence protocol, but the following is typical. Each core has a data cache for its own use. When a core writes data that other cores have cached, the cache coherence protocol forces the write to wait while the protocol finds the cached copies and invalidates them. When a core reads data that another core has just written, the cache coherence protocol doesn't return the data until it finds the cache that holds the modified data, annotates that cache to indicate there is a copy of the data, and fetches the data to the reading core. These operations take about the same time as loading data from off-chip RAM (hundreds of cycles), so sharing mutable data can have a disproportionate effect on performance.

Exercising the cache coherence machinery by modifying shared data can produce two kinds of scaling problems. First, the cache coherence protocol serializes modifications to the same cache line, which can prevent parallel speedup. Second, in extreme cases the protocol may saturate the inter-core interconnect, again preventing additional cores from providing additional performance. Thus good performance and scalability often demand that data be structured so that each item of mutable data is used by only one core.

In many cases scaling bottlenecks limit performance to some maximum, regardless of the number of cores. In other cases total throughput decreases as the number of cores grows, because each waiting core slows down the cores that are making progress. For example, non-scalable spin locks produce per-acquire interconnect traffic that is proportional to the number of waiting cores; this traffic may slow down the core that holds the lock by an amount proportional to the number of waiting cores [41]. Acquir-

ing a Linux spin lock takes a few cycles if the acquiring core was the previous lock holder, takes a few hundred cycles if another core last held the lock and there is no contention, and are not scalable under contention.

Performance is often the enemy of scaling. One way to achieve scalability is to use inefficient algorithms, so that each core busily computes and makes little use of shared resources such as locks. Conversely, increasing the efficiency of software often makes it less scalable, by increasing the fraction of time it uses shared resources. This effect occurred many times in our investigations of MOSBENCH application scalability.

Some scaling bottlenecks cannot easily be fixed, because the semantics of the shared resource require serial access. However, it is often the case that the implementation can be changed so that cores do not have to wait for each other. For example, in the stock Linux kernel the set of runnable threads is partitioned into mostly-private per-core scheduling queues; in the common case, each core only reads, writes, and locks its own queue [36]. Many scaling modifications to Linux follow this general pattern.

Many of our scaling modifications follow this same pattern, avoiding both contention for locks and contention for the underlying data. We solved other problems using well-known techniques such as lock-free protocols or fine-grained locking. In all cases we were able to eliminate scaling bottlenecks with only local changes to the kernel code. The following subsections explain our techniques.

4.2 Multicore packet processing

The Linux network stack connects different stages of packet processing with queues. A received packet typically passes through multiple queues before finally arriving at a per-socket queue, from which the application reads it with a system call like `read` or `accept`. Good performance with many cores and many independent network connections demands that each packet, queue, and connection be handled by just one core [21, 42]. This avoids inter-core cache misses and queue locking costs.

Recent Linux kernels take advantage of network cards with multiple hardware queues, such as Intel's 82599 10Gbit Ethernet (IXGBE) card, or use software techniques, such as Receive Packet Steering [26] and Receive Flow Steering [25], to attempt to achieve this property. With a multi-queue card, Linux can be configured to assign each hardware queue to a different core. Transmit scaling is then easy: Linux simply places outgoing packets on the hardware queue associated with the current core. For incoming packets, such network cards provide an interface to configure the hardware to enqueue incoming packets matching a particular criteria (e.g., source IP address and port number) on a specific queue and thus to a particular core. This spreads packet processing load across cores. However, the IXGBE driver goes further: for each core, it samples every 20^{th} outgoing TCP packet and updates the hardware's flow directing tables to deliver further incoming packets from that TCP connection directly to the core.

This design typically performs well for long-lived connections, but poorly for short ones. Because the technique is based on sampling, it is likely that the majority of packets on a given short connection will be misdirected, causing cache misses as Linux delivers to the socket on one core while the socket is used on another. Furthermore, because few packets are received per short-lived connection, misdirecting even the initial handshake packet of a connection imposes a significant cost.

For applications like Apache that simultaneously accept connections on all cores from the same listening socket, we address this problem by allowing the hardware to determine which core and thus which application thread will handle an incoming connection. We modify `accept` to prefer connections delivered to the local core's queue. Then, if the application processes the connection on the same core that accepted it (as in Apache), all processing for that connection will remain entirely on one core. Our solution has the added benefit of addressing contention on the lock that protects the single listening socket's connection backlog queue.

To implement this, we configured the IXGBE to direct each packet to a queue (and thus core) using a hash of the packet headers designed to deliver all of a connection's packets (including the TCP handshake packets) to the same core. We then modified the code that handles TCP connection setup requests to queue requests on a per-core backlog queue for the listening socket, so that a thread will accept and process connections that the IXGBE directs to the core running that thread. If `accept` finds the current core's backlog queue empty, it attempts to steal a connection request from a different core's queue. This arrangement provides high performance for short connections by processing each connection entirely on one core. If threads were to move from core to core while handling a single connection, a combination of this technique and the current sampling approach might be best.

4.3 Sloppy counters

Linux uses shared counters for reference-counted garbage collection and to manage various resources. These counters can become bottlenecks if many cores update them. In these cases lock-free atomic increment and decrement instructions do not help, because the coherence hardware serializes the operations on a given counter.

The MOSBENCH applications encountered bottlenecks from reference counts on directory entry objects (`dentry`s), mounted file system objects (`vfsmount`s), network routing table entries (`dst_entry`s), and counters

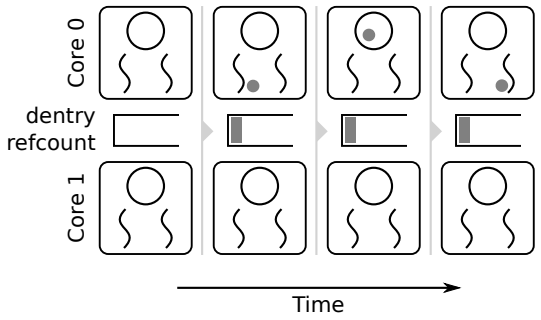

Figure 2: An example of the kernel using a sloppy counter for `dentry` reference counting. A large circle represents a local counter, and a gray dot represents a held reference. In this figure, a thread on core 0 first acquires a reference from the central counter. When the thread releases this reference, it adds the reference to the local counter. Finally, another thread on core 0 is able to acquire the spare reference without touching the central counter.

tracking the amount of memory allocated by each network protocol (such as TCP or UDP).

Our solution, which we call *sloppy counters*, builds on the intuition that each core can hold a few spare references to an object, in hopes that it can give ownership of these references to threads running on that core, without having to modify the global reference count. More concretely, a sloppy counter represents one logical counter as a single shared central counter and a set of per-core counts of spare references. When a core increments a sloppy counter by V, it first tries to acquire a spare reference by *decrementing* its per-core counter by V. If the per-core counter is greater than or equal to V, meaning there are sufficient local references, the decrement succeeds. Otherwise the core must acquire the references from the central counter, so it increments the shared counter by V. When a core decrements a sloppy counter by V, it releases these references as local spare references, *incrementing* its per-core counter by V. Figure 2 illustrates incrementing and decrementing a sloppy counter. If the local count grows above some threshold, spare references are released by decrementing both the per-core count and the central count.

Sloppy counters maintain the invariant that the sum of per-core counters and the number of resources in use equals the value in the shared counter. For example, a shared `dentry` reference counter equals the sum of the per-core counters and the number of references to the `dentry` currently in use.

A core usually updates a sloppy counter by modifying its per-core counter, an operation which typically only needs to touch data in the core's local cache (no waiting for locks or cache-coherence serialization).

We added sloppy counters to count references to `dentry`s, `vfsmount`s, and `dst_entry`s, and used sloppy counters to track the amount of memory allocated by each network protocol (such as TCP and UDP). Only uses of a counter that cause contention need to be modified, since sloppy counters are backwards-compatible with existing shared-counter code. The kernel code that creates a sloppy counter allocates the per-core counters. It is occasionally necessary to reconcile the central and per-core counters, for example when deciding whether an object can be de-allocated. This operation is expensive, so sloppy counters should only be used for objects that are relatively infrequently de-allocated.

Sloppy counters are similar to Scalable NonZero Indicators (SNZI) [22], distributed counters [9], and approximate counters [5]. All of these techniques speed up increment/decrement by use of per-core counters, and require significantly more work to find the true total value. Sloppy counters are attractive when one wishes to improve the performance of some uses of an existing counter without having to modify all points in the code where the counter is used. A limitation of sloppy counters is that they use space proportional to the number of cores.

4.4 Lock-free comparison

We found situations in which MOSBENCH applications were bottlenecked by low scalability for name lookups in the directory entry cache. The directory entry cache speeds up lookups by mapping a directory and a file name to a `dentry` identifying the target file's `inode`. When a potential `dentry` is located, the lookup code acquires a per-`dentry` spin lock to atomically compare several fields of the `dentry` with the arguments of the lookup function. Even though the directory cache has been optimized using RCU for scalability [40], the `dentry` spin lock for common parent directories, such as `/usr`, was sometimes a bottleneck even if the path names ultimately referred to different files.

We optimized `dentry` comparisons using a lock-free protocol similar to Linux' lock-free page cache lookup protocol [18]. The lock-free protocol uses a generation counter, which the PK kernel increments after every modification to a directory entry (e.g., `mv foo bar`). During a modification (when the `dentry` spin lock is held), PK temporarily sets the generation counter to 0. The PK kernel compares `dentry` fields to the arguments using the following procedure for atomicity:

- If the generation counter is 0, fall back to the locking protocol. Otherwise remember the value of the generation counter.
- Copy the fields of the `dentry` to local variables. If the generation afterwards differs from the remembered value, fall back to the locking protocol.
- Compare the copied fields to the arguments. If there is a match, increment the reference count unless it is 0, and return the `dentry`. If the reference count is 0, fall back to the locking protocol.

The lock-free protocol improves scalability because it allows cores to perform lookups for the same directory entries without serializing.

4.5 Per-core data structures

We encountered three kernel data structures that caused scaling bottlenecks due to lock contention: a per-super-block list of open files that determines whether a read-write file system can be remounted read-only, a table of mount points used during path lookup, and the pool of free packet buffers. Though each of these bottlenecks is caused by lock contention, bottlenecks would remain if we replaced the locks with finer grained locks or a lock free protocol, because multiple cores update the data structures. Therefore our solutions refactor the data structures so that in the common case each core uses different data.

We split the per-super-block list of open files into per-core lists. When a process opens a file the kernel locks the current core's list and adds the file. In most cases a process closes the file on the same core it opened it on. However, the process might have migrated to another core, in which case the file must be expensively removed from the list of the original core. When the kernel checks if a file system can be remounted read-only it must lock and scan all cores' lists.

We also added per-core `vfsmount` tables, each acting as a cache for a central `vfsmount` table. When the kernel needs to look up the `vfsmount` for a path, it first looks in the current core's table, then the central table. If the latter succeeds, the result is added to the per-core table.

Finally, the default Linux policy for machines with NUMA memory is to allocate packet buffers (`skbuff`s) from a single free list in the memory system closest to the I/O bus. This caused contention for the lock protecting the free list. We solved this using per-core free lists.

4.6 Eliminating false sharing

We found some MOSBENCH applications caused false sharing in the kernel. In the cases we identified, the kernel located a variable it updated often on the same cache line as a variable it read often. The result was that cores contended for the falsely shared line, limiting scalability. Exim per-core performance degraded because of false sharing of physical page reference counts and flags, which the kernel located on the same cache line of a `page` variable. memcached, Apache, and PostgreSQL faced similar false sharing problems with `net_device` and `device` variables. In all cases, placing the heavily modified data on a separate cache line improved scalability.

4.7 Avoiding unnecessary locking

For small numbers of cores, lock contention in Linux does not limit scalability for MOSBENCH applications. With more than 16 cores, the scalability of memcached, Apache, PostgreSQL, and Metis are limited by waiting for and acquiring spin locks and mutexes[1] in the file system and virtual memory management code. In many cases we were able to eliminate acquisitions of the locks altogether by modifying the code to detect special cases when acquiring the locks was unnecessary. In one case, we split a mutex protecting all the super page mappings into one mutex per mapping.

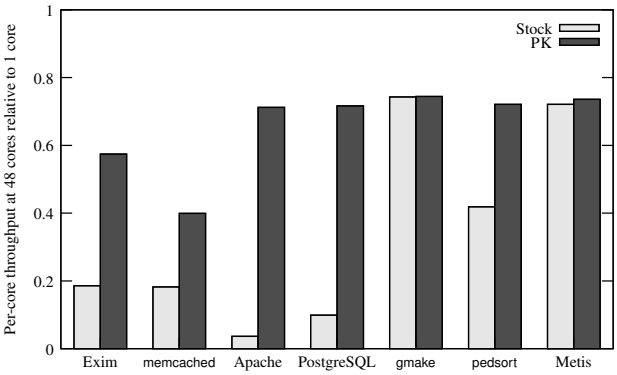

Figure 3: MOSBENCH results summary. Each bar shows the ratio of per-core throughput with 48 cores to throughput on one core, with 1.0 indicating perfect scalability. Each pair of bars corresponds to one application before and after our kernel and application modifications.

5 EVALUATION

This section evaluates the MOSBENCH applications on the most recent Linux kernel at the time of writing (Linux 2.6.35-rc5, released on July 12, 2010) and our modified version of this kernel, PK. For each application, we describe how the stock kernel limits scalability, and how we addressed the bottlenecks by modifying the application and taking advantage of the PK changes.

Figure 3 summarizes the results of the MOSBENCH benchmark, comparing application scalability before and after our modifications. A bar with height 1.0 indicates perfect scalability (48 cores yielding a speedup of 48). Most of the applications scale significantly better with our modifications. All of them fall short of perfect scalability even with those modifications. As the rest of this section explains, the remaining scalability bottlenecks are not the fault of the kernel. Instead, they are caused by non-parallelizable components in the application or underlying hardware: resources that the application's design requires it to share, imperfect load balance, or hardware bottlenecks such as the memory system or the network card. For this reason, we conclude that the Linux kernel with our modifications is consistent with MOSBENCH scalability up to 48 cores.

For each application we show scalability plots in the same format, which shows throughput per core (see, for example, Figure 4). A horizontal line indicates perfect

[1] A thread initially busy waits to acquire a mutex, but if the wait time is long the thread yields the CPU.

scalability: each core contributes the same amount of work regardless of the total number of cores. In practice one cannot expect a truly horizontal line: a single core usually performs disproportionately well because there is no inter-core sharing and because Linux uses a streamlined lock scheme with just one core, and the per-chip caches become less effective as more active cores share them. For most applications we see the stock kernel's line drop sharply because of kernel bottlenecks, and the PK line drop more modestly.

5.1 Method

We run the applications that modify files on a `tmpfs` in-memory file system to avoid waiting for disk I/O. The result is that MOSBENCH stresses the kernel more it would if it had to wait for the disk, but that the results are not representative of how the applications would perform in a real deployment. For example, a real mail server would probably be bottlenecked by the need to write each message durably to a hard disk. The purpose of these experiments is to evaluate the Linux kernel's multicore performance, using the applications to generate a reasonably realistic mix of system calls.

We run experiments on a 48-core machine, with a Tyan Thunder S4985 board and an M4985 quad CPU daughterboard. The machine has a total of eight 2.4 GHz 6-core AMD Opteron 8431 chips. Each core has private 64 Kbyte instruction and data caches, and a 512 Kbyte private L2 cache. The cores on each chip share a 6 Mbyte L3 cache, 1 Mbyte of which is used for the HT Assist probe filter [7]. Each chip has 8 Gbyte of local off-chip DRAM. A core can access its L1 cache in 3 cycles, its L2 cache in 14 cycles, and the shared on-chip L3 cache in 28 cycles. DRAM access latencies vary, from 122 cycles for a core to read from its local DRAM to 503 cycles for a core to read from the DRAM of the chip farthest from it on the interconnect. The machine has a dual-port Intel 82599 10Gbit Ethernet (IXGBE) card, though we use only one port for all experiments. That port connects to an Ethernet switch with a set of load-generating client machines.

Experiments that use fewer than 48 cores run with the other cores entirely disabled. memcached, Apache, Psearchy, and Metis pin threads to cores; the other applications do not. We run each experiment 3 times and show the best throughput, in order to filter out unrelated activity; we found the variation to be small.

5.2 Exim

To measure the performance of Exim 4.71, we configure Exim to use `tmpfs` for all mutable files—spool files, log files, and user mail files—and disable DNS and RFC1413 lookups. Clients run on the same machine as Exim. Each repeatedly opens an SMTP connection to Exim, sends 10 separate 20-byte messages to a local user, and closes the SMTP connection. Sending 10 messages per connection

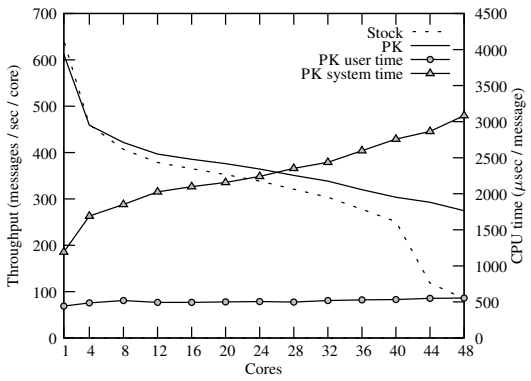

Figure 4: Exim throughput and runtime breakdown.

prevents exhaustion of TCP client port numbers. Each client sends to a different user to prevent contention on user mail files. We use 96 client processes regardless of the number of active cores; as long as there are enough clients to keep Exim busy, the number of clients has little effect on performance.

We modified and configured Exim to increase performance on both the stock and PK kernels:

- Berkeley DB v4.6 reads /proc/stat to find the number of cores. This consumed about 20% of the total runtime, so we modified Berkeley DB to aggressively cache this information.
- We configured Exim to split incoming queued messages across 62 spool directories, hashing by the per-connection process ID. This improves scalability because delivery processes are less likely to create files in the same directory, which decreases contention on the directory metadata in the kernel.
- We configured Exim to avoid an `exec()` per mail message, using `deliver_drop_privilege`.

Figure 4 shows the number of messages Exim can process per second on each core, as the number of cores varies. The stock and PK kernels perform nearly the same on one core. As the number of cores increases, the per-core throughput of the stock kernel eventually drops toward zero. The primary cause of the throughput drop is contention on a non-scalable kernel spin lock that serializes access to the `vfsmount` table. Exim causes the kernel to access the `vfsmount` table dozens of times for each message. Exim on PK scales significantly better, owing primarily to improvements to the `vfsmount` table (Section 4.5) and the changes to the `dentry` cache (Section 4.4).

Throughput on the PK kernel degrades from one to two cores, while the system time increases, because of the many kernel data structures that are not shared with one core but must be shared (with cache misses) with

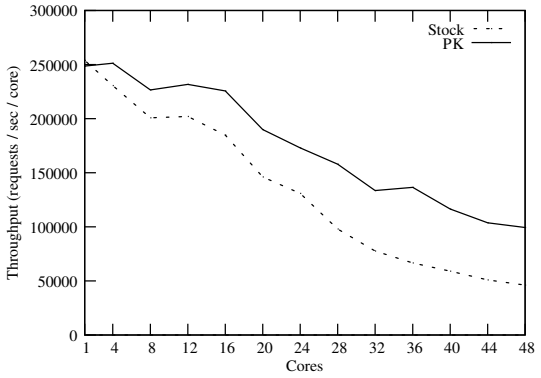

Figure 5: memcached throughput.

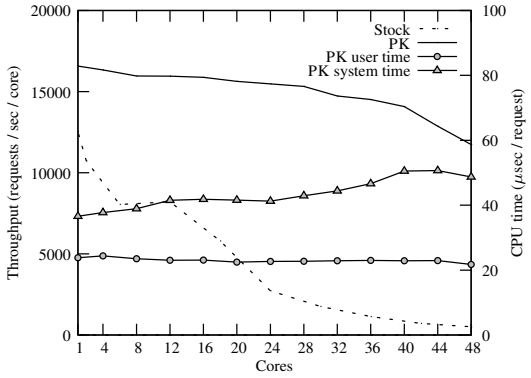

Figure 6: Apache throughput and runtime breakdown.

two cores. The throughput on the PK kernel continues to degrade; however, this is mainly due to application-induced contention on the per-directory locks protecting file creation in the spool directories. As the number of cores increases, there is an increasing probability that Exim processes running on different cores will choose the same spool directory, resulting in the observed contention.

We foresee a potential bottleneck on more cores due to cache misses when a per-connection process and the delivery process it forks run on different cores. When this happens the delivery process suffers caches misses when it first accesses kernel data—especially data related to virtual address mappings—that its parent initialized. The result is that process destruction, which frees virtual address mappings, and soft page fault handling, which reads virtual address mappings, execute more slowly with more cores. For the Exim configuration we use, however, this slow down is negligible compared to slow down that results from contention on spool directories.

5.3 memcached

We run a separate memcached 1.4.4 process on each core to avoid application lock contention. Each server is pinned to a separate core and has its own UDP port. Each client thread repeatedly queries a particular memcached instance for a non-existent key because this places higher load on the kernel than querying for existing keys. There are a total of 792 client threads running on 22 client machines. Requests are 68 bytes, and responses are 64. Each client thread sends a batch of 20 requests and waits for the responses, timing out after 100 ms in case packets are lost.

For both kernels, we use a separate hardware receive and transmit queue for each core and configure the IXGBE to inspect the port number in each incoming packet header, place the packet on the queue dedicated to the associated memcached's core, and deliver the receive interrupt to that core.

Figure 5 shows that memcached does not scale well on the stock Linux kernel.

One scaling problem occurs in the memory allocator. Linux associates a separate allocator with each socket to allocate memory from that chip's attached DRAM. The stock kernel allocates each packet from the socket nearest the PCI bus, resulting in contention on that socket's allocator. We modified the allocation policy to allocate from the local socket, which improved throughput by ∼30%.

Another bottleneck was false read/write sharing of IXGBE device driver data in the `net_device` and `device` structures, resulting in cache misses for all cores even on read-only fields. We rearranged both structures to isolate critical read-only members to their own cache lines. Removing a single falsely shared cache line in `net_device` increased throughput by 30% at 48 cores.

The final bottleneck was contention on the `dst_entry` structure's reference count in the network stack's destination cache, which we replaced with a sloppy counter (see Section 4.3).

The "PK" line in Figure 5 shows the scalability of memcached with these changes. The per core throughput drops off after 16 cores. We have isolated this bottleneck to the IXGBE card itself, which appears to handle fewer packets as the number of virtual queues increases. As a result, it fails to transmit packets at line rate even though there are always packets queued in the DMA rings.

To summarize, while memcached scales poorly, the bottlenecks caused by the Linux kernel were fixable and the remaining bottleneck lies in the hardware rather than in the Linux kernel.

5.4 Apache

A single instance of Apache running on stock Linux scales very poorly because of contention on a mutex protecting the single accept socket. Thus, for stock Linux, we run a separate instance of Apache per core with each server running on a distinct port. Figure 6 shows that Apache still scales poorly on the stock kernel, even with separate Apache instances.

For PK, we run a single instance of Apache 2.2.14 on one TCP port. Apache serves a single static file from an

ext3 file system; the file resides in the kernel buffer cache. We serve a file that is 300 bytes because transmitting a larger file exhausts the available 10 Gbit bandwidth at a low server core count. Each request involves accepting a TCP connection, opening the file, copying its content to a socket, and closing the file and socket; logging is disabled. We use 58 client processes running on 25 physical client machines (many clients are themselves multi-core). For each active server core, each client opens 2 TCP connections to the server at a time (so, for a 48-core server, each client opens 96 TCP connections).

All the problems and solutions described in Section 5.3 apply to Apache, as do the modifications to the `dentry` cache for both files and sockets described in Section 4. Apache forks off a process per core, pinning each new process to a different core. Each process dedicates a thread to accepting connections from the shared listening socket and thus, with the accept queue changes described in Section 4.2, each connection is accepted on the core it initially arrives on and all packet processing is performed local to that core. The PK numbers in Figure 6 are significantly better than Apache running on the stock kernel; however, Apache's throughput on PK does not scale linearly.

Past 36 cores, performance degrades because the network card cannot keep up with the increasing workload. Lack of work causes the server idle time to reach 18% at 48 cores. At 48 cores, the network card's internal diagnostic counters show that the card's internal receive packet FIFO overflows. These overflows occur even though the clients are sending a total of only 2 Gbits and 2.8 million packets per second when other independent tests have shown that the card can either receive upwards of 4 Gbits per second or process 5 million packets per second.

We created a microbenchmark that replicates the Apache network workload, but uses substantially less CPU time on the server. In the benchmark, the client machines send UDP packets as fast as possible to the server, which also responds with UDP packets. The packet mix is similar to that of the Apache benchmark. While the microbenchmark generates far more packets than the Apache clients, the network card ultimately delivers a similar number of packets per second as in the Apache benchmark and drops the rest. Thus, at high core counts, the network card is unable to deliver additional load to Apache, which limits its scalability.

5.5 PostgreSQL

We evaluate Linux's scalability running PostgreSQL 8.3.9 using both a 100% read workload and a 95%/5% read/write workload. The database consists of a single indexed 600 Mbyte table of 10,000,000 key-value pairs stored in `tmpfs`. We configure PostgreSQL to use a 2 Gbyte application-level cache because PostgreSQL protects its cache free-list with a single lock and thus

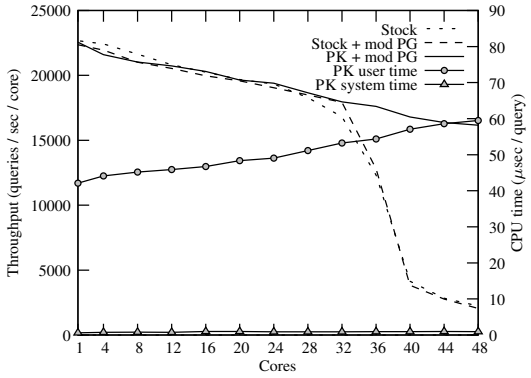

Figure 7: PostgreSQL read-only workload throughput and runtime breakdown.

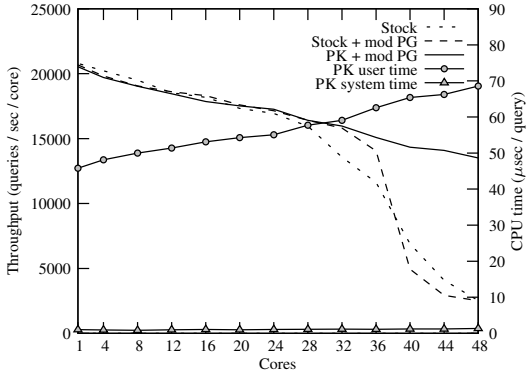

Figure 8: PostgreSQL read/write workload throughput and runtime breakdown.

scales poorly with smaller caches. While we do not pin the PostgreSQL processes to cores, we do rely on the IXGBE driver to route packets from long-lived connections directly to the cores processing those connections.

Our workload generator simulates typical high-performance PostgreSQL configurations, where middleware on the client machines aggregates multiple client connections into a small number of connections to the server. Our workload creates one PostgreSQL connection per server core and sends queries (selects or updates) in batches of 256, aggregating successive read-only transactions into single transactions. This workload is intended to minimize application-level contention within PostgreSQL in order to maximize the stress PostgreSQL places on the kernel.

The "Stock" line in Figures 7 and 8 shows that PostgreSQL has poor scalability on the stock kernel. The first bottleneck we encountered, which caused the read/write workload's total throughput to peak at only 28 cores, was due to PostgreSQL's design. PostgreSQL implements row- and table-level locks atop user-level mutexes; as a result, even a non-conflicting row- or table-level lock acquisition requires exclusively locking one of only 16 global mutexes. This leads to unnecessary contention for non-conflicting acquisitions of the same lock—as seen in

the read/write workload—and to false contention between unrelated locks that hash to the same exclusive mutex. We address this problem by rewriting PostgreSQL's row- and table-level lock manager and its mutexes to be lock-free in the uncontended case, and by increasing the number of mutexes from 16 to 1024.

The "Stock + mod PG" line in Figures 7 and 8 shows the results of this modification, demonstrating improved performance out to 36 cores for the read/write workload. While performance still collapses at high core counts, the cause of this has shifted from excessive user time to excessive system time. The read-only workload is largely unaffected by the modification as it makes little use of row- and table-level locks.

With modified PostgreSQL on stock Linux, throughput for both workloads collapses at 36 cores, with system time rising from 1.7 μseconds/query at 32 cores to 322 μseconds/query at 48 cores. The main reason is the kernel's `lseek` implementation. PostgreSQL calls `lseek` many times per query on the same two files, which in turn acquires a mutex on the corresponding `inode`. Linux's adaptive mutex implementation suffers from starvation under intense contention, resulting in poor performance. However, the mutex acquisition turns out not to be necessary, and PK eliminates it.

Figures 7 and 8 show that, with PK's modified `lseek` and smaller contributions from other PK changes, PostgreSQL performance no longer collapses. On PK, PostgreSQL's overall scalability is primarily limited by contention for the spin lock protecting the buffer cache page for the root of the table index. It spends little time in the kernel, and is not limited by Linux's performance.

5.6 gmake

We measure the performance of parallel gmake by building the object files of Linux 2.6.35-rc5 for x86_64. All input source files reside in the buffer cache, and the output files are written to `tmpfs`. We set the maximum number of concurrent jobs of gmake to twice the number of cores.

Figure 9 shows that gmake on 48 cores achieves excellent scalability, running 35 times faster on 48 cores than on one core for both the stock and PK kernels. The PK kernel shows slightly lower system time owing to the changes to the `dentry` cache. gmake scales imperfectly because of serial stages at the beginning of the build and straggling processes at the end.

gmake scales so well in part because much of the CPU time is in the compiler, which runs independently on each core. In addition, Linux kernel developers have thoroughly optimized kernel compilation, since it is of particular importance to them.

5.7 Psearchy/pedsort

Figure 10 shows the runtime for different versions of pedsort indexing the Linux 2.6.35-rc5 source tree, which

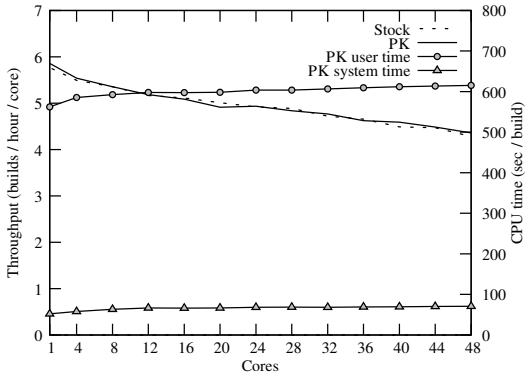

Figure 9: gmake throughput and runtime breakdown.

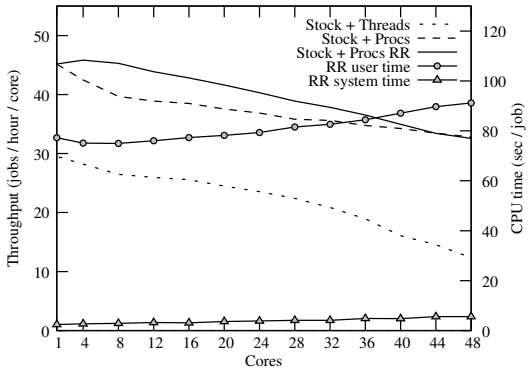

Figure 10: pedsort throughput and runtime breakdown.

consists of 368 Mbyte of text across 33,312 source files. The input files are in the buffer cache and the output files are written to `tmpfs`. Each core uses a 48 Mbyte word hash table and limits the size of each output index to 200,000 entries (see Section 3.6). As a result, the total work performed by pedsort and its final output are independent of the number of cores involved.

The initial version of pedsort used a single process with one thread per core. The line marked "Stock + Threads" in Figure 10 shows that it scales badly. Most of the increase in runtime is in system time: for 1 core the system time is 2.3 seconds, while at 48 cores the total system time is 41 seconds.

Threaded pedsort scales poorly because a per-process kernel mutex serializes calls to `mmap` and `munmap` for a process' virtual address space. pedsort reads input files using `libc` file streams, which access file contents via `mmap`, resulting in contention over the shared address space, even though these memory-mapped files are logically private to each thread in pedsort. We avoided this problem by modifying pedsort to use one process per core for concurrency, eliminating the `mmap` contention by eliminating the shared address space. This modification involved changing about 10 lines of code in pedsort. The performance of this version on the stock kernel is shown as "Stock + Procs" in Figure 10. Even on a single core,

the multi-process version outperforms the threaded version because any use of threads forces glibc to use slower, thread-safe variants of various library functions.

With a small number of cores, the performance of the process version depends on how many cores share the per-socket L3 caches. Figure 10's "Stock + Procs" line shows performance when the active cores are spread over few sockets, while the "Stock + Procs RR" shows performance when the active cores are spread evenly over sockets. As corroborated by hardware performance counters, the latter scheme provides higher performance because each new socket provides access to more total L3 cache space.

Using processes, system time remains small, so the kernel is not a limiting factor. Rather, as the number of cores increases, pedsort spends more time in the glibc sorting function `msort_with_tmp`, which causes the decreasing throughput and rising user time in Figure 10. As the number of cores increases and the total working set size per socket grows, `msort_with_tmp` experiences higher L3 cache miss rates. However, despite its memory demands, `msort_with_tmp` never reaches the DRAM bandwidth limit. Thus, pedsort is bottlenecked by cache capacity.

5.8 Metis

We measured Metis performance by building an inverted index from a 2 Gbyte in-memory file. As for Psearchy, we spread the active cores across sockets and thus have access to the machine's full L3 cache space at 8 cores.

The "Stock + 4 KB pages" line in Figure 11 shows Metis' original performance. As the number of cores increases, the per-core performance of Metis decreases. Metis allocates memory with `mmap`, which adds the new memory to a region list but defers modifying page tables. When a fault occurs on a new mapping, the kernel locks the entire region list with a read lock. When many concurrent faults occur on different cores, the lock itself becomes a bottleneck, because acquiring it even in read mode involves modifying shared lock state.

We avoided this problem by mapping memory with 2 Mbyte *super-pages*, rather than 4 Kbyte pages, using Linux's `hugetlbfs`. This results in many fewer page faults and less contention on the region list lock. We also used finer-grained locking in place of a global mutex that serialized super-page faults. The "PK + 2MB pages" line in Figure 11 shows that use of super-pages increases performance and significantly reduces system time.

With super-pages, the time spent in the kernel becomes negligible and Metis' scalability is limited primarily by the DRAM bandwidth required by the reduce phase. This phase is particularly memory-intensive and, at 48 cores, accesses DRAM at 50.0 Gbyte/second, just shy of the maximum achievable throughput of 51.5 Gbyte/second measured by our microbenchmarks.

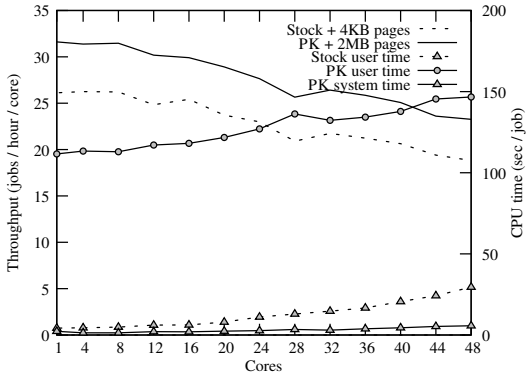

Figure 11: Metis throughput and runtime breakdown.

Application	Bottleneck
Exim	App: Contention on spool directories
memcached	HW: Transmit queues on NIC
Apache	HW: Receive queues on NIC
PostgreSQL	App: Application-level spin lock
gmake	App: Serial stages and stragglers
pedsort	HW: Cache capacity
Metis	HW: DRAM throughput

Figure 12: Summary of the current bottlenecks in MOSBENCH, attributed either to hardware (HW) or application structure (App).

5.9 Evaluation summary

Figure 3 summarized the significant scalability improvements resulting from our changes. Figure 12 summarizes the bottlenecks that limit further scalability of MOSBENCH applications. In each case, the application is bottlenecked by either shared hardware resources or application-internal scalability limits. None are limited by Linux-induced bottlenecks.

6 DISCUSSION

The results from the previous section show that the MOSBENCH applications can scale well to 48 cores, with modest changes to the applications and to the Linux kernel. Different applications or more cores are certain to reveal more bottlenecks, just as we encountered bottlenecks at 48 cores that were not important at 24 cores. For example, the costs of thread and process creation seem likely to grow with more cores in the case where parent and child are on different cores. Given our experience scaling Linux to 48 cores, we speculate that fixing bottlenecks in the kernel as the number of cores increases will also require relatively modest changes to the application or to the Linux kernel. Perhaps a more difficult problem is addressing bottlenecks in applications, or ones where application performance is not bottlenecked by CPU cycles, but by some other hardware resource, such as DRAM bandwidth.

Section 5 focused on scalability as a way to increase performance by exploiting more hardware, but it is usually also possible to increase performance by exploiting

a fixed amount of hardware more efficiently. Techniques that a number of recent multicore research operating systems have introduced (such as address ranges, dedicating cores to functions, shared memory for inter-core message passing, assigning data structures carefully to on-chip caches, etc. [11, 15, 53]) could apply equally well to Linux, improving its absolute performance and benefiting certain applications. In future work, we would like to explore such techniques in Linux.

One benefit of using Linux for multicore research is that it comes with many applications and has a large developer community that is continuously improving it. However, there are downsides too. For example, if future processors don't provide high-performance cache coherence, Linux's shared-memory-intensive design may be an impediment to performance.

7 Conclusion

This paper analyzes the scaling behavior of a traditional operating system (Linux 2.6.35-rc5) on a 48-core computer with a set of applications that are designed for parallel execution and use kernel services. We find that we can remove most kernel bottlenecks that the applications stress by modifying the applications or kernel slightly. Except for sloppy counters, most of our changes are applications of standard parallel programming techniques. Although our study has a number of limitations (e.g., real application deployments may be bottlenecked by I/O), the results suggest that traditional kernel designs may be compatible with achieving scalability on multicore computers. The MOSBENCH applications are publicly available at `http://pdos.csail.mit.edu/mosbench/`, so that future work can investigate this hypothesis further.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Brad Chen, for their feedback. This work was partially supported by Quanta Computer and NSF through award numbers 0834415 and 0915164. Silas Boyd-Wickizer is partially supported by a Microsoft Research Fellowship. Yandong Mao is partially supported by a Jacobs Presidential Fellowship. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

References

[1] Apache HTTP Server, May 2010. `http://httpd.apache.org/`.

[2] Exim, May 2010. `http://www.exim.org/`.

[3] Memcached, May 2010. `http://memcached.org/`.

[4] PostreSQL, May 2010. `http://www.postgresql.org/`.

[5] The search for fast, scalable counters, May 2010. `http://lwn.net/Articles/170003/`.

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

[7] AMD, Inc. Six-core AMD opteron processor features. `http://www.amd.com/us/products/server/processors/six-core-opteron/Pages/six-core-opteron-key-architectural-features.aspx`.

[8] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. of the 13th SOSP*, pages 95–109, 1991.

[9] 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.

[10] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[11] A. Baumann, P. Barham, P.-E. Dagand, T. Haris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In *Proc of the 22nd SOSP*, Big Sky, MT, USA, Oct 2009.

[12] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, 1990.

[13] D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *Computer*, 23(5):35–43, 1990.

[14] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for NUMA memory management. In *Proc. of the 12th SOSP*, pages 19–31, New York, NY, USA, 1989. ACM.

[15] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. D. Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proc. of the 8th OSDI*, December 2008.

[16] 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.

[17] B. Cantrill and J. Bonwick. Real-world concurrency. *Commun. ACM*, 51(11):34–39, 2008.

[18] J. Corbet. The lockless page cache, May 2010. `http://lwn.net/Articles/291826/`.

[19] A. L. Cox and R. J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with platinum. In *Proc. of the 12th SOSP*, pages 32–44, 1989.

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

[21] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc of the 22nd SOSP*, Big Sky, MT, USA, Oct 2009.

[22] F. Ellen, Y. Lev, V. Luchango, and M. Moir. SNZI: Scalable nonzero indicators. In *PODC 2007*, Portland, Oregon, USA, Aug. 2007.

[23] GNU Make, May 2010. `http://www.gnu.org/software/make/`.

[24] 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.

[25] T. Herbert. rfs: receive flow steering, September 2010. `http://lwn.net/Articles/381955/`.

[26] T. Herbert. rps: receive packet steering, September 2010. `http://lwn.net/Articles/361440/`.

[27] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[28] J. Jackson. Multicore requires OS rework Windows architect advises. *PCWorld magazine*, 2010. `http://www.pcworld.com/businesscenter/article/191914/multicore_requires_os_rework_windows_architect_advises.html`.

[29] Z. Jia, Z. Liang, and Y. Dai. Scalability evaluation and optimization of multi-core SIP proxy server. In *Proc. of the 37th ICPP*, pages 43–50, 2008.

[30] A. R. Karlin, K. Li, M. S. Manasse, and S. S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proc. of the 13th SOSP*, pages 41–55, 1991.

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

[32] A. Kleen. Linux multi-core scalability. In *Proceedings of Linux Kongress*, October 2009.

[33] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proc. of the 21st ISCA*, pages 302–313, 1994.

[34] R. P. LaRowe, Jr., C. S. Ellis, and L. S. Kaplan. The robustness of NUMA memory management. In *Proc. of the 13th SOSP*, pages 137–151, 1991.

[35] J. Li, B. T. Loo, J. M. Hellerstein, M. F. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *Proc. of the 2nd IPTPS*, Berkeley, CA, February 2003.

[36] Linux 2.6.35-rc5 source, July 2010. `Documentation/scheduler/sched-design-CFS.txt`.

[37] Linux kernel mailing list, May 2010. `http://kerneltrap.org/node/8059`.

[38] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT, 2010.

[39] 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.

[40] P. E. McKenney, D. Sarma, and M. Soni. Scaling dcache with rcu, Jan. 2004. `http://www.linuxjournal.com/article/7124`.

[41] 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.

[42] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *Proc. of the 1st OSDI*, page 10, Berkeley, CA, USA, 1994. USENIX Association.

[43] D. Patterson. The parallel revolution has started: are you part of the solution or the prolem? In *USENIX ATEC*, 2008. `www.usenix.org/event/usenix08/tech/slides/patterson.pdf`.

[44] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the ACM EuroSys Conference (EuroSys 2010)*, Paris, France, April 2010.

[45] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor system. In *Proceedings of HPCA*. IEEE Computer Society, 2007.

[46] C. Schimmel. *UNIX systems for modern architectures: symmetric multiprocessing and caching for kernel programmers*. Addison-Wesley, 1994.

[47] M. D. Schroeder and M. Burrows. Performance of Firefly RPC. In *Proc. of the 12th SOSP*, pages 83–90, 1989.

[48] J. Stribling, J. Li, I. G. Councill, M. F. Kaashoek, and R. Morris. Overcite: A distributed, cooperative citeseer. In *Proc. of the 3rd NSDI*, San Jose, CA, May 2006.

[49] J. H. Tseng, H. Yu, S. Nagar, N. Dubey, H. Franke, P. Pattnaik, H. Inoue, and T. Nakatani. Performance studies of commercial workloads on a multi-core system. *IEEE Workload Characterization Symposium*, pages 57–65, 2007.

[50] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proc. of the 13th SOSP*, pages 26–40, 1991.

[51] 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.

[52] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proc. of the 7th ASPLOS*, pages 279–289, New York, NY, USA, 1996. ACM.

[53] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.

[54] C. Yan, Y. Chen, and S. Yuanchun. Parallel scalability comparison of commodity operating systems on large scale multi-cores. In *Proceedings of the workshop on the interaction between Operating Systems and Computer Architecture (WIOSCA 2009)*.

[55] C. Yan, Y. Chen, and S. Yuanchun. OSMark: A benchmark suite for understanding parallel scalability of operating systems on large scale multi-cores. In *2009 2nd International Conference on Computer Science and Information Technology*, pages 313–317, 2009.

[56] C. Yan, Y. Chen, and S. Yuanchun. Scaling OLTP applications on commodity multi-core platforms. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 134–143, 2010.

[57] M. Young, A. Tevanian, R. F. Rashid, D. B. Golub, J. L. Eppinger, J. Chew, W. J. Bolosky, D. L. Black, and R. V. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc. of the 11th SOSP*, pages 63–76, 1987.