Multicore processor architectures have become ubiquitous in today’s computing platforms, especially in parallel computing installations, with their power and cost advantages. While the technology trend continues towards having hundreds of cores on a chip in the foreseeable future, an urgent question posed to system designers as well as application users is whether applications can receive sufficient support on today’s operating systems for them to scale to many cores. To this end, people need to understand the strengths and weaknesses on their support on scalability and to identify major bottlenecks limiting the scalability, if any. As open-source operating systems are of particular interests in the research and industry communities, in this paper we choose three operating systems (Linux, Solaris and FreeBSD) to systematically evaluate and compare their scalability by using a set of highly-focused microbenchmarks for broad and detailed understanding their scalability on an AMD 32-core system. We use system profiling tools and analyze kernel source codes to find out the root cause of each observed scalability bottleneck. Our results reveal that there is no single operating system among the three standing out on all system aspects, though some system(s) can prevail on some of the system aspects. For example, Linux outperforms Solaris and FreeBSD significantly for file-descriptor- and process-intensive operations. For applications with intensive sockets creation and deletion operations, Solaris leads FreeBSD, which scales better than Linux. With the help of performance tools and source code instrumentation and analysis, we find that synchronization primitives protecting shared data structures in the kernels are the major bottleneck limiting system scalability.

**key words:** operating systems, multicore, scalability, microbenchmarks

### 1. Introduction

Multicore architectures have been widely adopted by major microprocessor manufacturers including Intel, AMD, and Sun Microsystems. Computers with dual-cores or quad-cores processors are common and Intel 80-core prototype chip has been announced [1]. In the foreseeable future we would see processors with tens of even hundreds of cores on a single chip. With increasingly abundant parallelism available, an unavoidable question is whether it can be well exploited by applications, or whether the applications can scale to increasingly large number of cores. The success of multiple-core technology largely depends on scalability of applications running on the parallel platform. The scalability of an application not only is determined by its inherent parallelism, but also highly relies on the support of operating systems on which it runs. Many applications, especially those intensive on system services such as I/O, networking, as well as shared kernel data structures (process table, directory entry, and semaphore, etc.), might hardly benefit from the parallelism in the processor if the operating system serializes its services requested simultaneously by multiple processes. That is, the support of operating system on the scalability of applications, or in short, the scalability of operating system, in a multicore system is crucial in unleashing potentially high aggregate computing power of multicore processors and translating it into real application performance improvements.

While the scalability issue of operating systems has been studied in context of multiprocessor system (e.g., SMP machine) [2]–[4], the issue on multicore system is unique. First, the number of cores in multicore systems can be much larger than the number of processors in a multiprocessor system, which makes the scalability an even acute issue. Second, the cores on a chip usually share performance-critical hardware resources such as the last level on-chip cache and memory controller. Thus, for application users and system administrators using multicore systems, it is important to understand the distinction on scalability of different operating systems with different kernel-intensive applications. So that they can choose an operating system appropriate for their applications. For practitioners who aim to improve operating systems’ scalability and applications designers who want to avoid weakness on an operating system’s scalability support, identification of where the scalability issues, and how the issues are raised, is important and relies on insightful analysis of system behaviors and their correlations with corresponding kernel source codes.

To this end, we selected three widely used open-source operating systems, namely Linux, Solaris, and FreeBSD, for comparison. The availability of the source codes of these systems allows us to pinpoint root causes of experimentally observed scalability problems down to specific kernel functions and data structures, which is especially valuable for system designers. To understand how different components of an operating system contribute to the scalability problem of real-life applications, we designed a suite of highly focused microbenchmarks, each stressing one system components, including process creation and termination, memory mapping and unmapping, operations on file descriptors, socket creation and deletion, and System V IPC operations. Each of the microbenchmarks generates an increasing number of processes running on an increasing number of cores (up to 32 cores) to allow us to observe how the relevant kernel component(s) respond to the increasing concurrency im-
posed by user program and increasing parallelism provided by the hardware. By using system profiling tools, we can correlate the observed scalability bottlenecks to individual system calls, functions, and data structures in the source codes and reveal how a particular bottleneck is produced as well as the significance of the bottlenecks’ impact on the application’s performance. In summary, we made two major contributions.

- We designed a suite of well-targeted microbenchmarks to evaluate the scalability of three popular open-source operating systems and identified the scalability bottlenecks, serving as a reference for users to choose right operating systems and architectural scales for maximum cost-effectiveness of applications.
- With profiling tools and kernel code analysis we pinpoint the bottlenecks to the kernel functions and data structures, facilitating system designers to remove or alleviate scalability bottlenecks by code optimization.

2. Experimental Setup

2.1 Multicore Architecture and Operating Systems

We chose three open-source operating systems, which are Linux 2.6.26.8, Solaris (OpenSolaris 2008.11), and FreeBSD 8.0-CURRENT. Note that although newer versions exist for these three kernels, the experimental results and analysis reported here still make sense because most of bottlenecks found in this paper are not removed in newer versions [5]. All the three systems are UNIX or UNIX-like operating systems and are POSIX-compatible, or providing the same set of application program interfaces (API) for user programs to access system services. This allows us to run the benchmarks on each of the systems without any customizations, making the comparison not subject to evaluators’ personal choices.

Our experiments were performed on a 32-core NUMA system. In total there are eight AMD Opteron 8347HE chips, each with four cores running at 1.9 GHZ. Each core has a 32 KB L1 data cache and a 32 KB L1 instruction cache, both of which are 2-way associative. Each core has one L2 512 KB 8-way set associative cache. There is also a 2 MB 32-way set associative L3 cache shared by the four cores on each chip. Intra-chip cores are connected with an internal crossbar and all chips are connected with the HyperTransport interconnect. In the system there are 32 GB memory, which is evenly partitioned into eight banks, each directly connected one of the eight chips. There are three 1 TB hard disks, each being used to install one of the operating systems. Figure 1 illustrates the architecture of the multicore system.

2.2 Microbenchmarks

The microbenchmark suite has five programs (forkbench, mmapbench, dupbench, sembench and sockbench), each generating workload to stress one component of an operating system (process management, memory management, file descriptor management, System V IPC and network, respectively). In each of the benchmarks, there is a master process responsible for creating a number of worker processes, each assigned to a different core for generating system workload. We make sure that there are not any scalability issues existing in the user-level code, such as uses of lock or barrier, in the benchmarks. That is to say, in a load balanced running scenario any detected scalability deficiency must be caused by the kernel codes. The performance is measured as system throughput, which is defined as the number of system-related operations completed in one unit of time. As we always keep the same number of worker processes as that of tested cores, and each worker generates the same amount
of workload, the system scalability for a benchmark can be characterized as the change of throughput with the number of tested cores. In a fully scalable system, the throughput does not change as the workload increases at the same rate as the number of cores. If a scalability bottleneck exists, which can serialize the system services to simultaneous requests from the parallel workload, certain processes may not be able to make progress from time to time and the throughput can be degraded. The larger the degradation, the worse the system scalability.

2.3 Tracking Programs’ Executions

To identify and analyze scalability bottlenecks, it is not sufficient to know only the changes of throughput. Instead, we need to track the system behaviors related to potential scalability bottlenecks, such as the use of locks. To this end, we adopted various performance tools. Specifically, for Linux we use Oprofile [6] to acquire the function call graph of a program’s run and execution time of each concerned function, and use /proc/lock_stat [7] to identify highly contended locks. For Solaris, we use lockstat [8], a Dtrace [9] tool, to obtain the call graph, function execution times, and lock contentions. However, for FreeBSD, we get lock contention information only by lock profiling. Two popular kernel profiling tools in FreeBSD, PMC [10] and Kgmon [11], cannot work properly on our platform as they have not yet supported AMD Opteron CPUs. To avoid the impact of the profiling tools on the performance measurements such as throughput, we use the kernels compiled without the profiling capacity enabled for benchmarking and collecting performance data. We then enable the tools for profiling and tracking the system services.

To ensure the programs run in a load balanced manner with minimal overhead of process scheduling, we try to manually pin each process on a designated core. In Linux, we use the sched_setaffinity() system call for this purpose. We use pset_bind() and cpuset_setaffinity() for this purpose in Solaris and FreeBSD, respectively. Note that we cannot create a one-to-one mapping between processes and cores on Solaris if the number of processes is equal to 32 because at least one core is required to be unbound [12]. Therefore, we leave one core unbound when there are 32 processes. All benchmarks are compiled using GCC with the optimization level O3.

3. Identification of Scalability Bottlenecks

3.1 Forkbench

Forkbench is a microbenchmark stressing process management in an operating system. In the implementation, each worker process repeatedly creates a child process using fork() and then waits for its termination using waitpid() in a tight loop. The child terminates itself using exit() immediately when it gets scheduled. In this way, the system services on process creation and termination are intensively requested. The throughputs of the benchmark on different number of cores (from 1 to 32) on the three tested operating systems are shown in Fig. 2.

As indicated in the figure, Linux scales much better than FreeBSD and Solaris. While with single core the throughput is 4329 operations per second, or ops/s, for Linux, only 5 times and 3 times higher than Solaris and FreeBSD, respectively, the throughput of Linux is 26 times and 12 times higher than Solaris and FreeBSD on 32 cores. To get insights on the different throughput degradation, we show the breakdown of the total execution time of all worker processes for different operating systems with different number of cores in Fig. 3. As we can see from the figure, with the increase of cores, the CPU idle time takes rapidly increasing proportion over the total execution time for Solaris and FreeBSD. For example, with one core, the ratios for idle time are 0, and keep increasing to 70.0% and 18.3% with 32 cores for Solaris and FreeBSD, respectively. As user code is fully scalable, this clearly shows that some system services have been serialized leaving some CPUs idle.
Table 1

| Cores | Functions     | Percentage |
|-------|---------------|------------|
|       | Solaris       |            |
| 32    | mach_cpu_idle | 73%        |
|       | mutex_delay_default | 8%        |
|       | mutex_enter   | 3%         |
|       | (usermode)    |            |
|       | mutex_enter   | 34%        |
|       | hmutex_compare| 7%         |
| 1     |               |            |
|       | mutex_enter   | 6%         |
|       |               |            |
|       | Linux         |            |
| 32    | dup_mm        | 18%        |
|       | unlink_file_vma| 18%        |
|       | page_fault    | 9%         |
| 1     |               |            |
|       | page_fault    | 23%        |
|       | handle_mm_fault| 10%        |
|       | unmmap_vmas   | 3%         |

Even Linux is not fully immune to this effect, as we can see that the proportion for the kernel time is substantially increased, indicating that it takes longer time to complete the same number of system operations.

Let us first examine the bottlenecks with Solaris, which has the worst scalability among the three. After we sorted its kernel functions by their increased execution times per system operation, we found that `mach_cpu_idle()`, `mutex_delay_default()`, and `mutex_enter()` are three functions whose execution times increase the most with the increase of cores. Three hottest functions and their percentage over total execution times are also presented in Table 1. These results hint that certain system lock(s) cause the scalability problem. We then use the `lockstat` tool to identify the most contended lock, which turns out to be a read-write lock that is frequently called by function `zfs_getpage()`. Further studies of the profiling statistics reveal that the frequent calls of the function with the increased number of cores is due to frequent occurrences of minor page faults during the benchmark's execution (e.g., there are about 120,000 such faults per second with 32 cores). Our runtime call stack sampling indicates that function `zfs_getpage()` is called to handle the page faults. To find why such a large number of page faults are generated, we use the `Dtrace` tool to track virtual address of each page fault associated with acquisition of the read-write lock. Positioning these addresses into the address space map obtained by using the `pmap` tool [13], we know the page faults happen mostly when accessing text segment of `libc.so`, `ld.so`, and text and data segments of the benchmark's executable file.

In the operating system, when a new process is forked and ready to run, the library files (`libc` and `ld`) and the executable file have to be mapped into the process’s address space. While the libraries are dynamically linked to the program, all processes probably access the unique copy of a library file at runtime. Furthermore, the executable file is shared when worker processes and their child processes are forked. Thus, when the program starts and one of its processes accesses an address in the libraries or the executable file, the address is still invalid in the process’s page table and a page fault occurs accordingly to get corresponding pages from the shared files. As the files usually have been cached in memory, the page faults are only minor ones without asking for real I/O [14]. However, simultaneously incurring minor page faults does need to access lock-protected shared files. In addition, unmapping of the linked files when the processes terminate also needs to access the locks protecting the shared files. With severe contention on them, these locks become a scalability bottleneck with process creation and termination.

In spite of different severity in terms of throughput degradation, the bottlenecks for the benchmark are generated with a similar mechanism in FreeBSD and Linux. In FreeBSD, there are about 13,000 minor page faults in one second with 32 cores. Profiling of lock usage finds that the mutex lock protecting kernel data structure `virtual memory objects` [15] is heavily contended because of frequent calls by functions `vm_fault()`, `vm_object_deallocatem()`, and `vm_map_entry_delete()`. In FreeBSD, in addition to the text segments, access of data segments library files (`libc.so` and `ld.so`) and anonymous memory also causes minor page faults. In Linux, the locks protecting dynamically linked library files and program executable file are competed by functions `unlink_file_vma()` and `dup_mmap()`. The percentage of aggregate execution times of the two functions over program’s execution time is 1% with one core, but is increased to 36% on 32 cores (see Table 1).

To verify our analysis, we statically link the libraries to the benchmark, which removes the lock contention associated with `libc.so` and `ld.so`. We then run the benchmark in FreeBSD and Linux and the throughputs are shown in Fig. 2. We do not test Solaris as the capability of linking library statically is not supported by OpenSolaris 2008.11 [13]. After this minor modification, we do see the performance of the benchmark gets improved across the number of cores on Linux and FreeBSD.

The impact of lock contention on the system scalability differs vastly across the three operating systems because of the differences on their lock implementations. Linux uses spin lock to protect the shared memory-mapped file. Thus, as the lock becomes increasingly contention with the increase of cores, idle time does not increase and only system time gets increased due to busy waiting in the kernel mode. In contrast, Solaris uses read-write locks. A process is forced to sleep when it tries to acquire a busy read-write lock, leaving its assigned core idle. Similarly, FreeBSD uses mutex lock for the same purpose [15].

3.2 Mmapbench

In the mmapbench benchmark, each worker process runs a loop. In each of the loop’s iterations, a 500MB file is mapped onto a segment of the process’s address space with the `MAP_SHARED` flag. The entire mapped file is then read page by page, followed by an operation to destroy the mapping. Thus, the benchmark is to stress the component of an operating system for managing memory mapped files and reveal its performance and scalability, which are important...
for database and large-scale web server applications, where many accessed files are first mapped into memory [16].

Figure 4 shows the throughputs of mmapbench with different number of cores on the three operating systems. In the figure, Solaris scales better than Linux and FreeBSD until the number of cores reaches 29, though throughputs on Linux and FreeBSD are higher than Solaris when the number of cores is relatively small. Our measurements of CPU idle times in Fig. 5 show that the percentage of idle time over total execution time on Solaris remains at less than 2.0% before the number of cores increases to 29 and suddenly rises up to 50.3% after that. This suggests that some serialization caused by system contention limits Solaris’s scalability.

We had suspected that the contention is related to page faults when the mapped file gets accessed. Surprisingly we found few page faults with the running of the benchmark on any of the three systems. One possible reason is that the systems have adopted optimizations like pre-fault [17]. We then resort to profiling of kernel function execution times.

Table 2 Top 3 hot functions when running mmapbench.

| Cores | Functions                  | Percentage |
|-------|----------------------------|------------|
|       | `mach_cpu_idle` (usermode) | 51%        |
|       | `mutex_delay_default`      | 34%        |
|       | `mach_cpu_idle` (usermode) | 6%         |
| 32    | `unlock_file_vma`          | 97%        |
|       | `vma_link` (no symbols)    | 3%         |
|       | `vma_link`                 | 0%         |
| 32    | `unmap_vmas`               | 50%        |
|       | `page_fault`               | 46%        |
|       | `unlock_file_vma`          | 8%         |
|       | `page_fault`               | 5%         |

On FreeBSD, the story is similar. The vnode data structure, each associated with one file, is frequently accessed by several kernel functions (`vget()`, `vput()`, `vrele()`, `vref()`, `ufs_markatime()`, and `ufs_getattr()`) [15]. When a file is mapped or unmapped, the statistics, such as access time, reference count, attribute that are maintained in the file’s vnode, are looked up or updated. As vnode is protected by a mutex lock, it can become a scalability bottleneck with programs of intensive file mapping operations, such as the mmapbench benchmark.

An interesting observation on FreeBSD is that there is little CPU idle time no matter how many cores are used for the execution of mmapbench though the mutex lock protecting vnode could sleep a process requesting for it [15]. By analyzing the implementation of mutex lock in FreeBSD, including code in function `mtx_lock_sleep()`, we find that it has been optimized to improve lock utilization. When a process (process A) requests for a lock that is currently held by another process (process B), process A does not sleep if process B is in the running status, expecting the lock would be released soon. Only when process B is sleeping, process A goes to sleep. In the running of mmapbench, the number of worker processes equals to the number of cores so that the process holding the lock is always in the running status. So no process would go to sleep because of requesting the mutex lock. This is different from the scenario where `link_file_vma()`, which are called with mapping and unmapping of files (using functions `mmap()` and `munmap()`), respectively, whose execution times increase the most per system operation with the increase of cores. For example, their time percentages over total execution times with one core are 0.4% and 0.3%, but increase to 46% and 50%, respectively, with 32 cores (See Table 2). `vma_link()` and `unlock_file_vma()` are called for linking and unlinking virtual memory areas and need to access the `address_space` data structure, each associated with one file, in Linux [18]. The data structure is protected by a spin lock, which has to be acquired before these two functions can operate on the data structure. Thus, the spin lock contention intensifies when many concurrent processes request for the same lock to map and unmapp the same file.
forkbench runs on systems (see Fig. 3). In forkbench, each worker process in forkbench repeatedly creates a child process, causing the total number of processes to be larger than the number of cores. Accordingly, a process holding a lock can be scheduled into sleeping status, increasing lock holding time and causing other process requesting the lock to sleep.

For Solaris, we are particularly interested in its dramatic scalability deterioration when the number of cores reaches 29. We used the lockstat tool and found that a read-write lock requested by function lgrp_shm_policy_set() becomes disproportionately hot with the increase of cores. For example, the number of unsuccessful lock requests is 192 with 15 cores, or 13 for one core on average. However, the number increases to 1,571,217 with 32 cores, or 49,101 for one core on average. Furthermore, the hottest functions on Solaris is mach_cpu_idle() according to Table 2, which suggests that this read-write lock is contended so heavily that CPUs cannot do any useful work. While lgrp_shm_policy_set() is responsible for setting policy of physical memory allocation to exploit access locality in NUMA machines [13], it seems not to be directly related to the file mapping operations. By studying the function call graph and kernel code, we found that in Solaris each file is associated with a vnode, like in FreeBSD, and each vnode contains a field describing its memory allocation policy. When a file is mapped into memory by a process, the process will set the policy by writing to vnode. As one common file is repeatedly mapped by multiple processes, the read-write lock protecting the field serializes the writing operations. While the lock contention is modest with smaller number of cores and its performance impact is negligible, the probability of lock contention increases significantly when more cores are added, which dramatically degrades throughput.

3.3 Dupbench

We design dupbench to stress the file descriptor management of operating systems to reveal potential scalability bottleneck. In the benchmark, each worker process repeatedly duplicates its private file descriptor and then closes the duplicated one. The throughputs of the benchmark on the three operating systems are presented in Fig. 6. As shown in the figure, Linux is fully scalable for the benchmark, while the throughput curves of Solaris and FreeBSD drop continuously with the number of cores, indicating their poor scalability. The execution time breakdowns in Fig. 7 indicate that some operations in the kernel mode cause the poor scalability of Solaris and FreeBSD because the execution time percentages in kernel increase with the number of cores.

To find the bottlenecks limiting the scalability of FreeBSD, we profiled the use of locks but did not find locks that become very hot corresponding to the increase of cores. In addition, there are not any locks that are consistently ranked as the hottest ones with the increase of cores. An exhaustive search of code and profiling data reveals that the witness module used for avoiding deadlock is the root cause of the FreeBSD’s poor scalability. In FreeBSD, witness tracks all lock acquisition and releasing activities. When a thread requests for a lock, witness scans two linked lists to determine whether this lock can be granted safely [15]. In the running of the benchmark, although no single lock is identified as highly contended, the overhead of running witness increases with the increasing number of cores and can become a scalability bottleneck. To confirm our finding, we recompiled the FreeBSD kernel without the witness module, as it is a default option in FreeBSD 8.0-CURRENT. The throughputs for this modified kernel are presented in Fig. 6, which shows that FreeBSD scales as well as Linux, fully eliminating the bottleneck.

On Solaris, an adaptive mutex lock [13] (flock_lock), which is called in function flk_get_lock_graph(), becomes increasingly contended with the increase of cores. The hottest functions in Table 3 also verify the contention of locks. In the benchmark it is the closing of a file descriptor that leads to the calling of the function. When a worker process executes close() on a file descriptor, the POSIX semantic requires that all locks of the file must be cleaned. In Solaris
Table 3  Top 3 hot functions when running dupbench.

| Cores | Functions                | Percentage |
|-------|--------------------------|------------|
| 32    | mutex_delay              | 51%        |
|       | default                  |            |
|       | default_lock_delay       | 9%         |
|       | mach_cpu_idle            | 9%         |
| 1     | mach_cpu_idle            | 95%        |
|       | mach_cpu_pause           | 2%         |
|       | mutex_enter              | 0%         |

Linux

| Cores | Functions       | Percentage |
|-------|-----------------|------------|
| 32    | system_call     | 17%        |
|       | dupfd           | 14%        |
|       | sysret_check    | 11%        |
| 1     | system_call     | 6%         |
|       | page_fault      | 6%         |
|       | dupfd           | 4%         |

Fig. 8  The throughput of the sembench benchmark with different number of cores. Throughput is defined as number of completed system operations, which is the System V IPC operation, in one second, and is averaged over the tested cores.

this is an operation on the vnode of the file by ZFS, Solaris’s default file system. ZFS organizes the lock information of all vnodess in a global hash table, which is protected by flock_lock. Although processes request operations on different vnodess, they all finally need to acquire the lock for the hash table, making it become a scalability bottleneck.

3.4 Sembench

Sembench is a microbenchmark designed to reveal the scalability of System V IPC in operating systems. In the benchmark, each worker process and its child process operate on a pair of System V semaphores in the ping-pong mode. To make the benchmark run correctly on 32 cores in FreeBSD, we set kernel options kern.ipc.semmni and kern.ipc.semmns to 64 and 100, respectively.

The throughputs and execution time breakdowns of the benchmark on the three operating systems are presented in Fig. 8 and Fig. 9, respectively, which show that Linux scales the worst. In the meantime, FreeBSD and Solaris can be fully scaled. To get the insights on the Linux’s disappointing scalability, we profiled the function execution times and found that functions _down_read() and _up_read() have their execution times increased the most per system operation with the increase of cores. As shown in Table 4, when there are 32 cores, more than 90% of total execution time is spent on these functions (47% for _down_read() and 47% for _up_read()). Both of two functions acquire a read lock protecting a global semaphore associated with a certain type of System V IPC resource such as message queue, semaphore, and shared memory. In the benchmark, although different worker processes operate on different System V semaphores, acquiring the read lock on the same read-write semaphore concurrently contends on the semaphore’s spin lock, creating a scalability bottleneck.

3.5 Sockbench

The sockbench benchmark is designed to evaluate the scalability of networking with socket operations. In the benchmark, each worker process repeatedly calls functions socket() and close(). Figure 10 shows throughputs of the benchmark on the three operating systems. As it shows, all operating systems have scalability bottlenecks. Among the three systems, Solaris scales better than FreeBSD, which does better than Linux. Figure 11 presents the execution time breakdowns of three operating systems. As suggested in the figure, when executing sockbench, most of the execu-

Table 4  Top 3 hot functions when running sembench.

| Cores | Functions       | Percentage |
|-------|-----------------|------------|
| 32    | (usermode)      | 27%        |
|       | mach_cpu_idle   | 18%        |
|       | cpu_pause       | 11%        |
| 1     | (usermode)      | 95%        |
|       | mutex_enter     | 1%         |

Linux

| Cores | Functions       | Percentage |
|-------|-----------------|------------|
| 32    | _down_read      | 47%        |
|       | _up_read        | 47%        |
|       | (no symbols)    | 2%         |
| 1     | main            | 16%        |
|       | semop           | 8%         |
|       | _d_lookupup     | 6%         |
Fig. 10 The throughput of the sockbench benchmark with different number of cores. Throughput is defined as number of completed system operations, either opening a socket or closing a socket, in one second, and is averaged over the tested cores.

Fig. 11 Breakdown of total execution time of all worker processes for sockbench on different operating systems with different number of cores according to processor status: being idle, executing kernel code, or executing user code.

Table 5 Top 3 hot functions when running sockbench.

| Cores | Functions                  | Percentage |
|-------|----------------------------|------------|
| 32 Solaris       | mutex_delay_default        | 52%        |
|       | default_lock_delay         | 10%        |
|       | mutex_enter                | 9%         |
| 1 Linux         | kmem_cache_alloc           | 26%        |
|       | tcp_open                   | 6%         |
|       | d_alloc                    | 33%        |
|       | d_instantiate              | 33%        |
|       | atomic_dec_and_lock        | 28%        |
|       | tick_do_update_jiffies64   | 14%        |
|       | memset_c                   | 10%        |
|       | kmem_cache_free            | 4%         |

tion time are spent in the kernel mode for all OSEs.

On Linux, there are three kernel functions, namely, d_alloc(), d_instantiate(), and atomic_dec_and_lock(), increasing their execution times the most per system operation with the increase of cores. On 32 cores, as shown in Table 5, more than 90% of total execution time is spent on these functions. The first two functions are used to allocate and initialize dentry instances, and the third function is called when a dentry or an inode instance is deleted. In the Linux kernel, each socket is associated with a dentry and an inode. When opening or closing a socket, the corresponding dentry and inode data structures are allocated or deleted from a global dentry cache and a inode linked list, respectively. However, the cache and the list are protected by locks, dcache_lock and inode_lock, respectively. It is the contention on the two locks that limits the scalability of the benchmark on Linux.

Different from Linux, Solaris and FreeBSD scale poorly because of operations in the network protocol stack. When a socket is created or deleted, a TCP stream between the application and the network device driver needs to be established or destroyed[13] in Solaris. The routines for establishing or destroying the stream need to update a reference count in the data structure for the protocol stack, which is shared by all streams with the same protocol type. Thus, intensive contention occurs with frequent socket opening and closing, causing a scalability bottleneck. The scalability bottleneck in FreeBSD is generated similarly due to network stack sharing. In the FreeBSD kernel, a network protocol is represented by an inpcbinfo data structure. This structure links all protocol control blocks[15] of a protocol and is protected by a read-write lock. When a socket is created, a new protocol control block is inserted into the linked list. When a socket is closed, corresponding protocol control block is modified to reflect its current status or removed from the linked list. However, the read-write lock protecting the list can be highly contended due to intensive socket operations in the benchmark.

3.6 Summary of Microbenchmark Evaluation

Table 6 summarizes the scalability of the five microbenchmarks on the three operating systems with a brief description of their respective bottlenecks. Although different bottlenecks can be introduced on different kernels for the same benchmark due to different kernel implementations, most bottlenecks are caused by synchronization primitives protecting shared data structures in the kernels.

4. Discussion

4.1 Bottleneck

In this paper, we conclude that the synchronization primitives protecting shared data structures are the root cause of poor kernel scalability, but we do not remove all identified shared data structures in kernels to verify this conclusion because doing so may need to redesign several subsystems of an operating system, which is challenging and time-consuming. However, our conclusion remains convincing because of two facts. First, our methodology of identifying the scalability bottleneck has been sufficiently verified. Recall that
we find bottlenecks by calculating the execution time increments of functions per system operation when switching 1 core to 32 cores. If the increment of a function is relatively large, this function is problematic in scalability and should be analyzed first. The methodology is proposed in our previous work [19] and the execution time increment of a function is called scalability value. Guided by this information, we had successfully identified and improved the scalability of two OLTP (Online Transaction Processing) applications on multicores. Second, there exists two examples that can support our conclusion. One is that linking library files statically in forbench makes the benchmark run faster across the number of cores. Remember that the scalability bottleneck of forbench is the contention of locks protecting shared files, including libraries and the executable file. Although linking libraries statically cannot remove all the sharing, we do see the improvement. The other example is that recent Linux kernel avoids to acquire the global read-write semaphore when locking an IPC resource in the kernel function ipc_lock(). This enhancement removes the kernel bottleneck found in sembench. We verify this enhancement by running sembench on the Linux 2.6.32 and the results are presented in Fig. 8 (labeled as Linux2.6.32). As we can see, the sembench scales almost perfectly on our platform, which is another evidence of the correctness of our conclusion.

4.2 Methodology of Choosing an OS

According to the evaluation results (without considering the results after scalability enhancements), we can see no single OS scales better than the others in all cases. This result implies that we should choose a proper OS on multicore systems according to the characteristics of an application. Specifically, if an application creates lots of new processes or file descriptors in a short time, Linux should be selected because it performs best in forbench and dupbench; if an application handles files by mmap, uses System V IPC heavily, or contains socket bombs, Solaris is the best choice because it outperforms Linux and FreeBSD in these aspects; if an application operates System V IPC heavily, we can also select FreeBSD as the OS because it scales as well as Solaris in sembench. Notice that the application characteristics used to select an OS can be represented by the frequencies of system calls and can be easily collected by the use of tools such as Dtrace, SystemTap and Strace.

4.3 Platform

This paper uses an AMD 32-core system to compare the scalability of three OSes. However, several recent proposals of multicore scalability [5], [20], [21] use an AMD 48-core platform for experiments. On the 48-core platform, we expect that bottlenecks found in this paper still exist, but the effect are even larger because the probability of contenting the same lock becomes higher. For the benchmarks that scale well on the AMD 32-core (e.g., dupbench in Linux), a new scalability bottleneck may appear on the 48-core system if the number of cores is large enough to make a lock become heavily contended (similar to mmapbench on Solaris with more than 29 cores).

5. Related Work

5.1 Comparison Study of Operating Systems

Researchers have extensively examined major operating systems from various aspects. Among the works, Lai and Baker evaluated the performance of three major UNIX kernels (Linux, Solaris, and FreeBSD) on the Pentium architecture by running a set of microbenchmarks and macrobenchmarks [22]. In the work, they conclude that no single operating system offers clearly better performance than others. In practice, it is often other non-technical factors such as ease of installation or freely available source code can be the reasons for choosing a particular one. Spinellis compared four operating systems (Linux, Solaris, FreeBSD, and Windows Research Kernel [23]) on file origination, code structure, code style, and the use of C preprocessor [24]. The paper reports that there are no significant code quality differences between the four systems although they went
through vastly different development processes. Bruning compared implementations of basic subsystems, including process scheduler, memory management system, and file system), of Linux, Solaris, and FreeBSD [25]. The author finds that these systems are similar at many aspects, such as time-shared scheduling and virtual file system layer. The author in [16] compared the algorithm scalability of basic network programming primitives, such as `socket()` and `bind()`, implemented in five kernels (Linux 2.4, Linux 2.6, FreeBSD, NetBSD, and OpenBSD). Our work is concerned with parallel scalability of different operating systems when they are stressed by multiple processes simultaneously and intensively requesting system services. Comparison study on this scalability issue becomes more relevant as multicore is becoming the mainstream architecture. By revealing different bottlenecks and their root causes in these three major open-source operating systems, our work complements previous comparison studies by providing another meaningful perspective to view these systems.

5.2 Research on Kernel Parallel Scalability

Gough et al examined the Linux kernel scalability using OLTP applications and found that contention on the lock protecting the run queue can significantly degrade the system scalability [26]. Veal and Foong ran a web server application on an Intel 8-core platform and found that front-side bus saturation is the root cause of the poor scalability [27]. Boyd-Wickizer et al evaluated Linux’s scalability by running seven applications on a 48-core machine. They reported a series of scalability bottlenecks and proposed their respective solutions [5]. Many of the identified bottlenecks in Linux are also confirmed in this work. In addition, this work studied two other operating systems and revealed their differences on the scalability issue.

To improve kernel scalability, some researchers propose to build new operating systems for multiprocessor architectures. For example, K42 [2] and Tornado [4], which are designed in the object-oriented manner, can achieve both locality and scalability on multiprocessors. However, the scalability is not yet known on large-scale multicores. In a recent research, Boyd-Wickizer et al show that almost all scalability problems are caused by the lock contention on shared data structures, as shown in the paper, and they implemented a prototype operating system for many cores in which applications are allowed to control how sharing is conducted[28]. Baumann et al proposed to maximize the kernel scalability by building a sharing-nothing OS [29], that is, each core owns its private OS copy. All cores communicate with each other by explicit messaging instead of shared memory. In [30], Wentzlaff and Agarwal suggest that the classical modules in the commodity operating systems (e.g., scheduler, memory management) should be factored to support many cores. In this work, we show the root cause of scalability by pinpointing to concrete data structure and kernel functions, which should help system designers in their efforts on the removal or amelioration of the bottlenecks.

6. Concluding Remarks

In this paper we investigate the scalability of three open-source operating systems on an AMD 32-core system using a suite of highly focused microbenchmarks. Evaluations on the systems show that no system scales clearly better than the others in all aspects. Performance data and related kernel source code analysis reveal that most of the scalability problems are caused by the kernel synchronization primitives used to protect shared data structures. Our efforts on comparing operating systems about their scalability and on revealing root causes of identified scalability bottlenecks through comprehensive benchmarking should be helpful for application users to select appropriate operating system and for system designers to solve system scalability issues.

References

[1] I.R. Advances, “’Era of Tera’ world’s first programmable processor to deliver teraflops performance with remarkable energy efficiency,” http://www.intel.com/pressroom/archive/releases/20070204comp.htm.
[2] 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 Transactions on Computer Systems (TOCS), vol.25, Issue 3, 2007.
[3] R. Bryant, J. Hawkes, and J. Steiner, “Scaling Linux to the extreme: From 64 to 512 processors,” Proc. Ottawa Linux Symposium, 2004.
[4] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, “Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system,” Proc. 3rd Symposium on Operating Systems Design and Implementation, pp.87–100, 1999.
[5] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pustrev, M.F. Kaashoek, R. Morris, and N. Zeldovich, “An analysis of linux scalability to many cores,” Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’10), Vancouver, Canada, Oct. 2010.
[6] “Oprofile-0.9.4,” http://sourceforge.net/projects/oprofile/.
[7] “Linux Kernel Source 2.6.25,” Documentation/lockstat.txt.
[8] “Solaris Lockstat,” http://docs.sun.com/app/docs/doc/819-2240/lockstat-1m?z=zh_TW&a=view.
[9] B.M. Cantrill, M.W. Shapiro, and L. Adam H, “Dynamic instrumentation of production systems,” Proc. USENIX 2004 Annual Technical Conference, 2004.
[10] “PMC Tutorial,” http://people.freebsd.org/~jkoshy/projects/perf-measurement/api-tutorial.html.
[11] R. Kinsella, “Profiling and Debugging the FreeBSD Kernel,” white paper, 321772-001, Intel Corporation, 2009.
[12] M. Laux, “Solaris Processor Sets Made Easy,” 2001. http://developers.sun.com/solaris/articles/solaris_processor.html.
[13] R. McDougall and J. Mauro, Solaris Internals Solaris 10 and OpenSolaris Kernel Architecture, Second ed., Prentice Hall, 2007.
[14] S. Cho and Y. Cho, “Page fault behavior and two prepagin schemes,” Conference Proceedings of the 1996 IEEE Fifteenth Annual International Phoenix Conference on Computers and Communications, pp.15–21, 1996.
[15] M.K. McKusick and G.V. Neville-Neil, “The Design and Implementation of the FreeBSD Operating System,” First ed., Addison-Wesley Professional, 2004.
[16] “Benchmarking BSD and Linux,” http://bulk.fefe.de/scalability/.
[17] U. Drepper, “Memory part 7: Memory Performance Tools,” http://lwn.net/Articles/257209/
[18] D.P. Bovet and M. Cesati, “Understanding Linux Kernel,” The Third Edition.

[19] Y. Cui, Y. Chen, and Y. Shi, “Scaling OLTP applications on commodity multi-cores,” Proc. 2010 IEEE International Symposium on Performance Analysis of Systems and Software, pp.134–143, 2010.

[20] T.I. Salomie, I.E. Subasu, J. Giceva, and G. Alonso, “Database engines on multicores, why parallelize when you can distribute?,” EuroSys, 2011.

[21] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang, “A case for scaling applications to many-core with OS clustering,” EuroSys, 2011.

[22] K. Lai and M. Baker, “A performance comparison of UNIX operating systems on the Pentium,” Proc. 1996 USENIX Conference, 1996.

[23] “Windows Research Kernel,” http://www.microsoft.com/resources/sharedsource/windowsacademic/researchkernelkit.mspx.

[24] D. Spinellis, “A tale of four kernels,” Proc. 30th International Conference on Software Engineering, pp.381–390, 2008.

[25] M. Bruning, “A comparison of Solaris, Linux and FreeBSD kernels,” http://opensolaris.org/os/article/2005-10-14a_comparison_of_solaris_linux_and_freebsd_kernel

[26] C. Gough, S. Siddha, and K. Chen, “Kernel Scalability—Expanding the Horizon Beyond Fine Grain Locks,” Proc. Linux Symposium, pp.153–165, 2007.

[27] B. Veal and A. Foong, “Performance scalability of a multi-core web server,” Proc. 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems, pp.57–66, New York, NY, USA, 2007.

[28] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, “Corey: An operating system for many cores,” Proc. 8th USENIX Symposium on Operating Systems Design and Implementation, 2008.

[29] A. Baumann, P. Barham, P.E. Dagand, T. Haris, R. Isaacs, S. Peter, A. Schupbach, and A. Singhania, “The multikernel: A new OS architecture for scalable multicore systems,” Conference Proceedings of SOSP 2009, 2009.

[30] D. Wentzlaff and A. Agarwal, “Factored operating systems (FOS): The case for a scalable operating system for multicores,” SIGOPS Operating System Review, pp.76–85, 2009.

Yan Cui received his Bachelor’s degree in computer science of Beijing University of Posts and Telecommunications in 2007. He is currently a Ph.D. candidate in Department of computer science and technology, Tsinghua University. His research focuses on the scalability of operating systems on multicore processors.

Yu Chen is a professor of the department of computer science, Tsinghua University. He received his Ph.D., M.S. and B.S. in computer science from National University of Defence Technology. His research focuses on operating systems and pervasive computing.

Yuanchun Shi is a professor of the Department of Computer Science, the director of HCI & Media Integration Institute, Tsinghua University. She received her Ph.D., M.S. and B.S. in Computer Science from Tsinghua University. She was a senior visiting scholar at MIT AI lab in 2001–2002. Her research interests include pervasive computing, human computer interaction, network multimedia and elearning technology.