OSMark: A Benchmark Suite for Understanding Parallel Scalability of Operating Systems on Large Scale Multi-Cores

Yan Cui† Yu Chen† Yuanchun Shi†
† *Tsinghua University*
Beijing, China
cui-y07@mails.tsinghua.edu.cn,{yuchen, shiyc}@tsinghua.edu.cn

Abstract

With the development of transistor technology, multi-core has become mainstream. Predictions based on Moore's Law state that a processor chip can accommodate thousands of cores in 5-10 years. As the system software between applications and hardware, can operating systems scale with the number of cores and achieve the performance potential?

In order to answer this question, a micro-benchmark suite OSMark is designed to understand the parallel scalability of operating systems on large scale multi-cores. Different from application-oriented benchmark, OSMark works by stressing separate parts of an operating system including process management, memory management, network, file descriptor operation and System V IPC. Evaluations on AMD 32-core machine with Linux as its OS indicate that most of benchmarks in OSMark scale bad. Linux kernel source code analysis and performance data reveal that kernel synchronization primitives protecting the shared data are the main bottlenecks limiting parallel scalability.

1. Introduction

Due to power and processor complexity of single-core design, chip manufactures such as Intel, AMD and Sun Microsystems have switched from single-core processors to dual-core or quad-core processors. Trends seen from Intel's 80-core prototype chip [1] suggest that cores in a single chip are still increasing and we are stepping into a new era where many-core or multi-core computing will be ubiquitous.

A natural consideration is that whether operating systems can make use of the computing power (cores) effectively and accelerate applications. Unfortunately, the result is poorly understood. In this paper, we design a micro-benchmark suite OSMark to reveal the parallel scalability of commodity operating systems on large scale multi-core platforms. Many frequently used primitives coming from the basic building blocks of commodity operating systems are benchmarked by OSMark. In addition, a unified benchmark framework and POSIX-compatible benchmark implementation make OSMark both modularization and portable.

978-1-4244-4520-2/09/$25.00 ©2009 IEEE

We evaluate OSMark on an AMD 32-core machine and use Linux as its operating system. Experimental results indicate that Linux has scalability problems in process creation/deletion, socket creation/deletion, memory mapped file creation/deletion and system V semaphore operation. The only test with good parallel scalability is the file descriptor operation. To understand the Linux scalability behavior, we analyze the related implementation of Linux kernel and exploit performance tools to identify scalability bottlenecks. Study results show that kinds of kernel synchronization primitives (i.e., spin lock, read-write lock and semaphore) which are used to protect shared data are the main bottlenecks preventing multiple tasks from progressing concurrently.

This paper makes three contributions: 1. We design a micro-benchmark suite OSMark to explore the parallel scalability of operating systems on multi-cores. 2. OSMark is evaluated on a real 32-core machine and a few scalability problems are found for Linux. 3. We provide thorough analysis for each benchmark to understand the root causes of the poor scalability.

The rest of papers are organized as follows: Section 2 describes the micro-benchmark suite OSMark. Section 3 introduces the multi-core platform and operating system for benchmarking OSMark. Section 4 evaluates the OSMark and analyzes bottlenecks affecting the scalability. Section 5 discusses the future work. Section 6 summarizes our conclusions.

2. OSMark Overview

Two kinds of processes exist in the implementation of OSMark: master process and slave process. The master process is responsible for launching each slave process at the same time and reporting the benchmark result, while the slave process performs the actual measurement. Each slave process is forked by the same master process and communicates with the master process with a unique pipe. The time interval of completing a workload concurrently is considered as the performance. In the implementation of OSMark, we acquire the time interval by timing the slave process which finishes its workload fastest. The parallel

scalability of an operating system is understood by changing the number of cores running the workload. Specifically, if the performance of OSMark running with *m* cores and *n* cores is the same, OSMark scales perfectly from *m* cores to *n* cores.

Forkbench, *mmapbench*, *dupbench*, *sockbench*, and *sembench* are five kinds of tests performed by OSMark. *Forkbench* stresses the process creation and deletion performance in the operating system, *mmapbench* focuses on memory mapped file creation and deletion scalability, *dupbench* benchmarks the file descriptor operation, *sockbench* measures the performance of socket creation and deletion, and *sembench* emphasizes on the basic System V IPC primitives. Each micro-benchmark in the OSMark is programmed with POSIX-compatible interfaces. Therefore, OSMark is portable between POSIX-compatible operating systems such as Linux, Solaris and BSD series.

In order to reduce the interference of OS scheduler, we bind each slave process to a separate computing core by the *sched_setaffinity()* system call. Furthermore, all micro-benchmarks are compiled using *gcc* with the optimization level O3.

3. Test Platform and OS

All experiments are carried out on an AMD 32-core NUMA machine with 32G memory. There are eight AMD Opteron chips in the system and each chip accommodates four cores. Each core owns its private L1 data cache, L1 instruction cache and L2 data cache. The L1 cache is 64Kbytes and 2-way set associative, while the L2 cache is 512Kbytes and 8-way set associative. Four cores on one chip share a unified L3 cache. The L3 cache is 2Mbytes and 32-way set associative. The frequency of each core is 1.9GHZ. Intra-chip cores and separate chips are connected by the HyperTransport [2] interconnection which has a frequency of 1GHZ. Memory is partitioned in eight banks, each connected to one of the eight chips. The architecture of the system is presented in Figure 1.

We use Ubuntu Linux with kernel version 2.6.26.8 as the operating system. To make results more accurate, several kernels are compiled and installed. Most of experimental results come from the clean kernel without debugging and statistics capabilities. Other kernels are used to collect debugging information or statistic information for lock.

4. Evaluations

We evaluate the OSMark benchmark suite on the platform described in section 3. Our evaluations are made up of five parts which will be discussed in the following subsections.

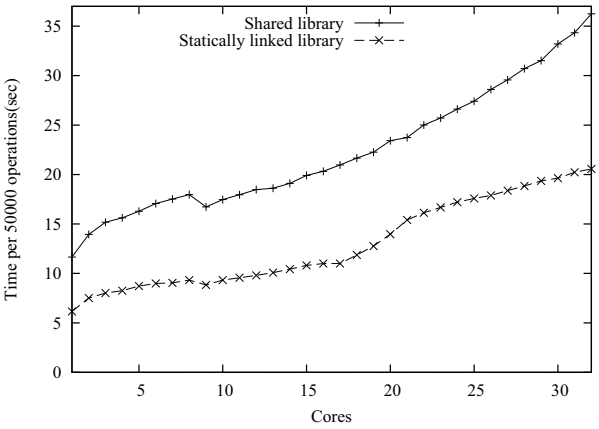

Figure 2. *Forkbench* scalability with increasing cores.

4.1. Process Creation and Deletion

The process management performance of an operating system is represented by *forkbench*. Each slave process in the implementation of *forkbench* calls *fork()* to create child process in a tight loop, while the child process just executes *exit()* system call when it is scheduled. The parallel scalability of *forkbench* is presented in Figure 2 (curve with dynamic). As indicated by this figure, the time of completing a workload for one core increases when there are more cores in the system. The trend of this curve suggests that the parallel scalability of *forkbench* is poor.

In order to understand the scalability behavior, *Oprofile* [3] is exploited to collect execution time information on the function basis. Experimental results on 32 cores show that *unlink_file_vma()* and *dup_mm()* are two functions which have largest execution time with 15.74% and 14.94% respectively. */proc/lock_stat* [4] is also used to explore lock contention when executing *forkbench*. *Copy_process()*, *do_exit()* and *do_wait()* are identified as contending for the *tasklist_lock* read-write lock, while *dup_mm()* and *unlink_file_vma()* contend for the same memory mapped file lock.

Related Linux kernel source code [5] is analyzed to understand the performance data. In the implementation of Linux kernel 2.6.26.8, each process is managed in three kinds of data structures. The first one is the family relationship which is organized in a tree data structure. Each process in the family tree has references pointing to its father, children and siblings. The second one is a double linked list which connects all processes together, while the third one is a hash table which is exploited to localize the *PCB* (process control block) according to the *PID* quickly. When a new process is created or deleted, its *PCB* will be maintained in the three data structures. However, in order to keep consistency, all of the three data structures are protected by the global read-

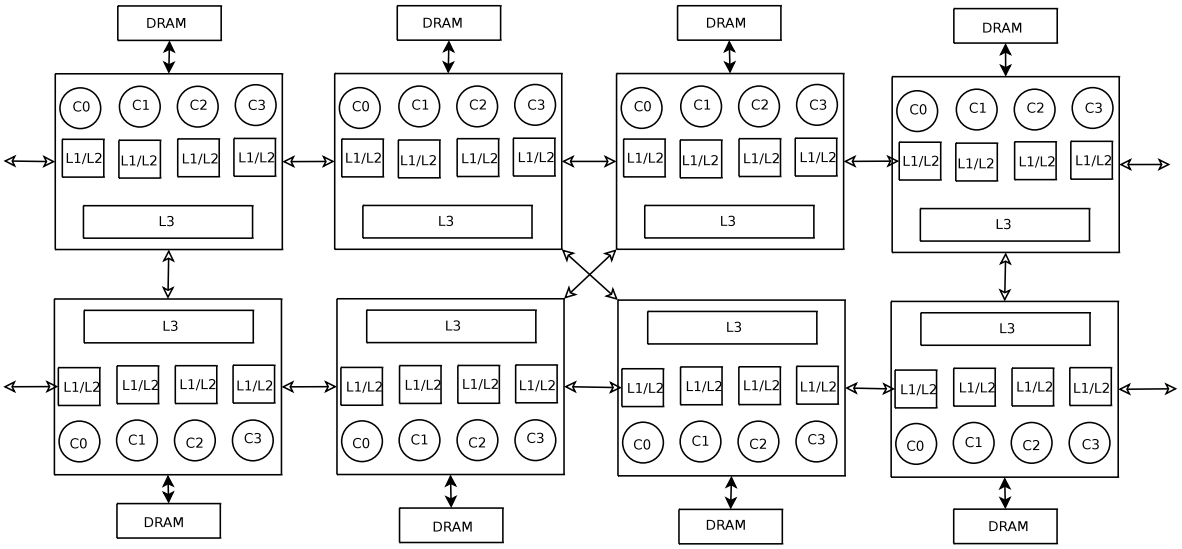

Figure 1. The architecture of AMD 32-core machine. Circles represent cores, squares with *L1/L2* represent data cache, L1 instruction cache and L2 data cache, rectangles with *L3* represent L3 unified cache, rectangles with *DRAM* represent memory, hollow arrows represent the HyperTranspost interconnection, and solid arrows represent memory bus.

write lock *tasklist_lock*. Thus, lock contention happens when multiple processes progress concurrently.

Another part that Linux kernel degrades the parallel scalability of *forkbench* is due to the sharing of memory mapped files. Although *forkbench* is designed to be concurrent, several memory mapped files are still shared by all child processes. This phenomenon can be identified by designing an experiment. The name of all shared memory mapped files are outputted by *printk()* in the Linux kernel when a process is created. Experimental results indicate that dynamic linker program (*ld-2.7.so*), libc library (*libc-2.7.so*) and the executable file *forkbench* are three shared memory mapped files. However, the first two sharings can be removed by linking the libraries statically. Figure 2 presents the result (statically linked library). As we can see, *forkbench* with statically linked libraries runs faster then *forkbench* with shared libraries, but it still scales poorly when using more cores in the system.

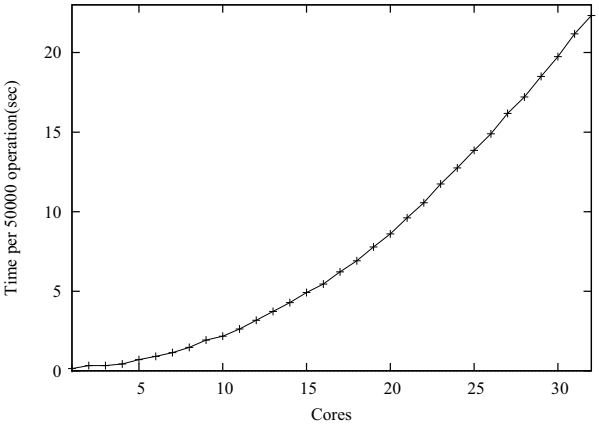

Figure 3. *Mmapbench* scalability with increasing cores.

4.2. Memory Mapped File Creation and Deletion

It is beneficial for databases and large web servers to map files into memory instead of having a buffer because more memory can be left to operating systems for I/O buffering [6]. Linux exports *mmap()* system call to map files into the memory [7]. We use *mmapbench* in OSMark to explore the parallel scalability of memory mapped file operation. Specifically, each slave process maps the same series of continuous pages of the same file, touches each page and destroys the mapping in a tight loop. In the experiments, the mapping is created with *MAP_SHARED* flag and we touch 125000 pages in a single mapping. Figure 3 presents the parallel scalability performance of *mmapbench*. As shown in the figure, *mmapbench* scales extremely bad when we increase cores in the system.

Performance data reveal that two functions *vma_link()* and *unlink_file_vma()* have the largest execution time (46.02% and 49.97%) and lock contention. Call-graph information and source code analysis show that the two functions are called when adding a virtual memory address range into the process address space or deleting a virtual memory address

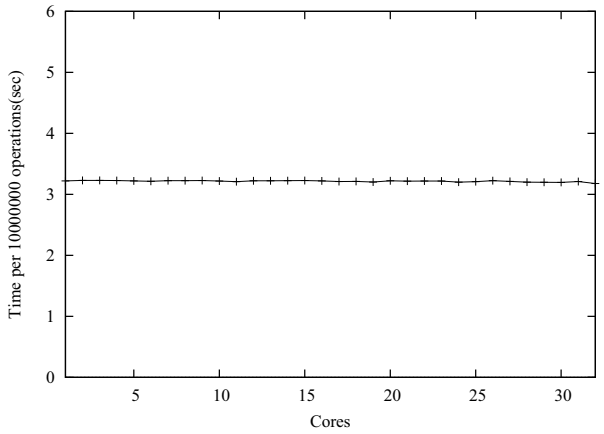

Figure 4. *Dupbench* scalability with increasing cores.

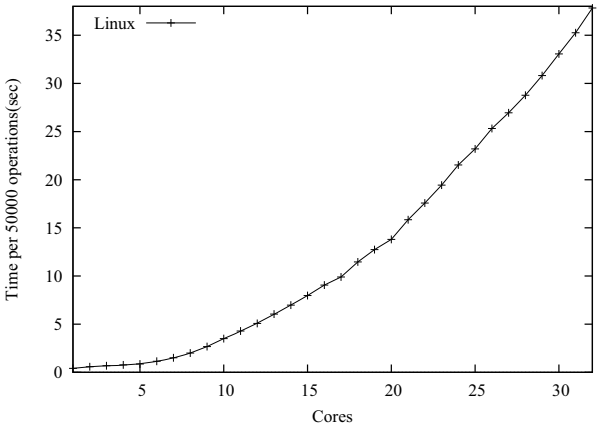

Figure 5. *Sockbench* scalability with increasing cores.

range. Memory mapped file sharing is the bottleneck affecting the parallel scalability of *mmapbench*. When multiple slave processes call *mmap()* or *unmap()* concurrently, the memory mapped file address range should be added into or deleted from each slave process address space. However, the same spin lock protecting the memory mapped file address range should be held or released. Thus, spin lock contention degrades the parallel scalability performance of the benchmark.

4.3. File Descriptor Operation

In this paper, *dupbench* is exploited to benchmark the performance of file descriptor operation. In the implementation, each slave process *dup()* and *close()* a private file descriptor in a tight loop. In the UNIX-like operating systems, *dup()* system call allocates the lowest unused file descriptor in the file resource of a process [8]. This test explores the parallel performance of *dup()*. The scalability result is shown in Figure 4. As indicated in this figure, *dupbench* scales rather well when we use more cores in the evaluation. Benchmarking other UNIX-like operating systems using *dupbench* is left as the future work. Part of experimental results on the version of Solaris 2008.11 [9] show that *dupbench* scales poorly when increasing the number of cores. We are now investigating this phenomenon.

4.4. Socket Creation and Deletion

The performance and scalability of socket is especially important to client/server and browser/server applications. In the socket parallel scalability test, each slave process in *sockbench* calls *socket()* and *close()* in a tight loop. The experimental result is presented in Figure 5.

Like *mmapbench*, *sockbench* also scales poorly. Time based sampling in *Oprofile* indicates that three functions *_atomic_dec_and_lock()*, *d_instantiate()* and *d_alloc()* have the largest execution time (35.02%, 25.46% and 25.32%). By understanding the implementation of *socket()* system call and *VFS* (virtual file system) layer in the Linux kernel, we find that each active socket is bound to a file descriptor, and each file descriptor is associated with an *inode* data structure and a *dentry* data structure. When creating a socket, a new *dentry* structure will be allocated and initialized. The functions *d_alloc()* and *d_instantiate()* are responsible for allocating a *dentry* instance in the *dentry* cache and initializing the instance using the *inode* information of the relative file descriptor respectively. Call-graph information suggests that the third function *_atomic_dec_and_lock()* which actually acquires a lock is called by deleting a *dentry* or *inode* data structure code path most of the time.

These functions degrade the parallel scalability of *sockbench* in two components. The first one is the spin lock contention of *dcache_lock*. This happens when multiple processes allocate, initialize and delete *dentry* data structure concurrently. In the Linux kernel, all *dentry* operations are preformed in the unique *dentry* cache after acquiring the global spin lock *dcache_lock*. The second one is the spin lock contention of *inode_lock*. In Linux, *VFS* organizes all *inodes* in a linked list which is protected by *inode_lock*. Thus, lock contention happens when multiple *inodes* are removed from the *inode* linked list simultaneously.

5. System V Semaphore Operation

To measure the performance of system V semaphore, we design the micro-benchmark *sembench* in which each slave process and its child process operate on a pair of system V semaphores in the ping-pong mode. We present the results in Figure 6.

As indicated in this figure, the *sembench* scales rather well when we use no more than five cores. However,

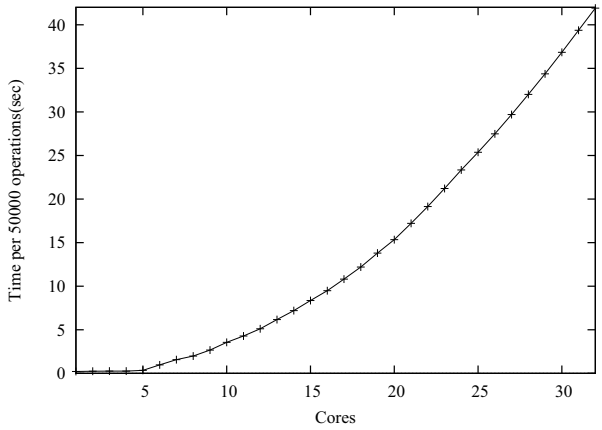

Figure 6. *Sembench* scalability with increasing cores.

when adding more cores in the system, the curve increases exponentially. Profile based on time sampling points out that two functions *__down_read()* and *__up_read()* are hottest when using 32 cores (46.6% and 46.58%). In addition, call-graph information shows that the two functions are called by *ipc_lock()* which is again called by *sys_semtimedop()*.

By analyzing the Linux kernel relative to system V IPC, we realize that each kind of system V IPC resource (shared memory, semaphore, and message queue) is protected by a global read-write semaphore in the *ipc_ids* data structure. Before operating on an instance of a system V IPC resource, which is represented by the *kern_ipc_perm* data structure, the process should acquire the semaphore first in the function *__down_read*. Contention for this read-write semaphore prevents multiple processes from progressing concurrently although each core operates on its private system V semaphore.

6. Discussion and Future Work

This paper evaluates the parallel scalability of Linux kernel 2.6.26.8 using OSMark and analyzes the bottlenecks degrading the scalability. The future work includes the following two parts:

- Our evaluations are carried out under Linux. However, it is not yet known whether other commodity operating systems have the same scalability problem. We will benchmark Solaris and FreeBSD [10] using OSMark.
- Many novel technologies such as RCU lock [11] (Read-Copy-Update) and scalable spin lock [12] have been proposed to improve the parallel scalability of operating systems on the multiprocessor platform. However, multi-core or many-core platforms impose new challenges to the OS. In the future, effective implementations of synchronization primitives will be explored to make operating systems run well on the many-core platform.

7. Conclusions

This paper targets for understanding the parallel scalability of commodity operating systems on the large scale multi-core platform. In order to achieve this goal, OSMark is designed to benchmark basic building blocks of a typical UNIX-like operating system including process creation/deletion, memory mapped file creation/deletion, file descriptor operation, socket creation/deletion and system V semaphore operation. Evaluations on the AMD 32-core system suggest that all micro-benchmarks in OSMark except *dupbench* scale bad. Kernel source analysis and novel experiments indicate that kernel synchronization primitives are the root causes of the poor scalability.

References

[1] I. R. Advances, "'era of tera' world's first programmable processor to deliver teraflops performance with remarkable energy efficiency." [Online]. Available: http://www.intel.com/pressroom/archive/releases/20070204comp.htm

[2] 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," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[3] "Oprofile-0.9.4." [Online]. Available: http://sourceforge.net/projects/oprofile/

[4] "Lockstat." [Online]. Available: http://www.kernel.org/

[5] "Linux kernel source." [Online]. Available: http://www.kernel.org/

[6] "Benchmarking bsd and linux," http://bulk.fefe.de/scalability/.

[7] D. P. Bovet and M. Cesati, "Understanding linux kernel," the Third Edition.

[8] W. Stevens, "Advanced programming in the unix environment."

[9] "Sun solaris." [Online]. Available: http://www.sun.com/software/solaris/

[10] "Freebsd project." [Online]. Available: http://www.freebsd.org/

[11] P. McKenney, D.Sarma, A.Arcangeli, A.Kleen, O.Krieger, and R. Russell, "Read-copy update," in *In Proceedings of the Linux Symposium 2002*, 2002, pp. 338–367.

[12] J. Mellor-Crummey and M.L.Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," vol. 9, no. 1, pp. 21–65, 1991.