FLSCHED: A Lockless and Lightweight Approach to OS Scheduler for Xeon Phi

Heeseung Jo
Chonbuk National University
567 Baekje-daero
Jeonju, Jeollabuk 54896
heeseung@jbnu.ac.kr

Woonhak Kang
Georgia Institute of Technology
266 Ferst Dr
Atlanta, GA 30313
woonhak.kang@gatech.edu

Changwoo Min
Virginia Tech
302 Whittemore
Blacksburg, VA 24060
changwoo@vt.edu

Taesoo Kim
Georgia Institute of Technology
266 Ferst Dr
Atlanta, GA 30313
taesoo@gatech.edu

ABSTRACT

Processor manufacturers have increased the number of cores in a chip, and the latest manycore processor has up to 76 physical cores and 304 hardware threads. On the other hand, the revolution of OS schedulers to manage processes in systems is slow to follow up emerging manycore processors.

In this paper, we show how much CFS, the default Linux scheduler, can break the performance of parallel applications on manycore processors (e.g., Intel Xeon Phi). Then, we propose a novel scheduler named FLSCHED, which is designed for lockless implementation with less context switches and more efficient scheduling decisions. In our evaluations on Xeon Phi, FLSCHED shows better performance than CFS up to 1.73× for HPC applications and 3.12× for micro-benchmarks.

ACM Reference format:
Heeseung Jo, Woonhak Kang, Changwoo Min, and Taesoo Kim. 2017. FLSCHED: A Lockless and Lightweight Approach to OS Scheduler for Xeon Phi. In *Proceedings of APSys '17, Mumbai, India, September 2, 2017,* 8 pages.
https://doi.org/10.1145/3124680.3124724

1 INTRODUCTION

Manycore processors are now prevalent in all types of computing devices, including mobile devices, servers, and hardware accelerators. For example, a single Xeon processor has up to 24 physical cores or 48 hardware threads [14], and a Xeon Phi processor has up to 76 physical cores or 304 hardware threads [22, 24]. In addition, due to increasingly important machine learning workloads, which are compute-intensive and massively parallel, we expect that the core count per system will increase further.

The prevalence of manycore processors imposes new challenges in scheduler design. First, schedulers should be able to handle the unprecedented high degree of parallelism. When the CFS scheduler was introduced, quad-core servers were dominant in data centers. Now, 32-core servers are standard in data centers [18]. Moreover, servers with more than 100 cores are becoming popular [2]. Under such a high degree of parallelism, a small sequential part in a system can break the performance and scalability of an application. Amdahl's Law says that if a sequential part in an entire system increases from 1% to 2%, then we end up with significantly decreased maximum speed up from 50 times to 33 times. In particular, schedulers in the Linux kernel use various lock primitives, such as spinlock, mutex, and read-write semaphore, to protect their data structures (see Table 1). We found that those sequential parts in schedulers protected by locks significantly degrade the performance of massively parallel applications. The performance degradation becomes especially significant in communication-intensive applications, which need scheduler intervention (see Figure 1 and Figure 2).

Second, the cost of context switching keeps increasing as the amount of context, which needs to be saved and restored, increasing. In Intel architectures, the width of SIMD register file has increased from 128-bits to 256-bits and now to 512-bits in XMM, YMM, and AVX, respectively [19]. This problem becomes exaggerated when the limited memory bandwidth is shared among many CPU cores with small cache such as Xeon Phi processors. Recent schedulers adopt

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

APSys '17, September 2, 2017, Mumbai, India

© 2017 Association for Computing Machinery.

ACM ISBN 978-1-4503-5197-3/17/09. . . $15.00
https://doi.org/10.1145/3124680.3124724

lazy optimization techniques, which do not save unchanged register files, to reduce the context switching overhead [8, 27]. However, there is no way but paying high cost for compute-intensive applications, which heavily rely on SIMD operations for better performance.

In this paper, we present FLSCHED—a new process scheduler to address the aforementioned problems. FLSCHED is designed for manycore accelerators like Xeon Phi. We adopt a lockless design to keep FLSCHED from becoming a sequential bottleneck. This is particularly critical for manycore accelerators, which have a large number of CPU cores; the Xeon Phi processor, which we used for experiment in this paper, has 57 cores or 228 hardware threads. FLSCHED is also designed for minimizing the number of context switches. Because a Xeon Phi processor has 2× larger vector registers than Xeon processors (i.e., 32 512-bit registers for Xeon Phi and 16 512-bits registers for Xeon processor), and its per-core memory bandwidth and cache size are smaller than a Xeon processor, its overhead of context switching is higher than a Xeon processor [9]. Thus, it is critical to minimize the number of context switching as many as possible. Finally, FLSCHED is tailored to throughput-oriented workloads, which are dominant in manycore accelerators.

This paper makes following three contributions:

- We evaluate how the widely-used Linux schedulers (i.e., CFS, FIFO, and RR) perform in a manycore accelerator. We analyze their behavior especially in terms of spinlock contention, which will increase a sequential portion in a scheduler, and the number of context switching.
- We design a new processor scheduler, named FLSCHED, which is tailored for minimizing the number of context switching in a lockless fashion.
- We show the effectiveness of FLSCHED for real-world OpenMP applications and micro-benchmarks. In particular, FLSCHED outperforms all other Linux schedulers for NAS Parallel Benchmark (NPB) by up to 73%.

The rest of this paper is organized as follows: §2 provides the motivation of our work with a case study, and §3 describes FLSCHED's design in detail. §4 evaluates and analyses FLSCHED's performance. §5 compares FLSCHED with previous research, and §6 concludes the paper.

2 CASE STUDY ON XEON PHI

In this section, we analyze how existing schedulers in the Linux kernel perform on manycore processors, especially a Xeon Phi processor. A lot of research efforts have been made to make OS scalable to fully utilize manycore processors [3–7, 12, 23]. Our focus in this paper is to investigate whether existing schedulers in the Linux kernel are efficient and scalable enough to manage manycore processors. To this end, we evaluated performance of three widely-used schedulers, CFS,

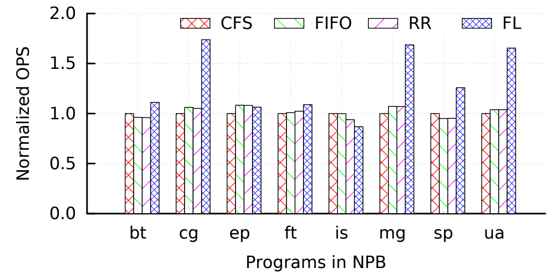

Figure 1: Performance comparison of NPB benchmarks running 1,600 threads on a Xeon Phi with different process schedulers. Performance (OPS: operations per second) is normalized to that of CFS.

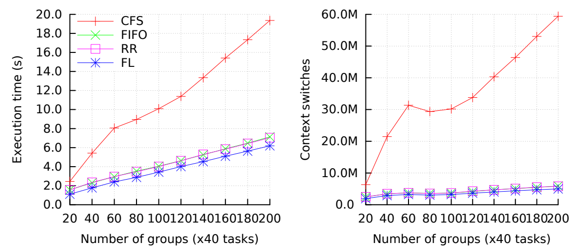

Figure 2: Execution time and number of context switches of hackbench on a Xeon Phi. We used the thread test mode with increasing the number of groups. Each group has 20 senders and 20 receivers communicating via pipe.

FIFO, and RR, with high performance computing (HPC) applications and a communication-intensive micro-benchmark.

We first measured the performance of NAS Parallel Benchmark (NPB) [1], which is written in OpenMP, running on the Xeon Phi. In particular, we ran eight NPB benchmarks, which fit in the Xeon Phi memory, and measured the operation per second (OPS). As Figure 1 shows, there is no clear winner among CFS, FIFO, and RR: for five benchmarks, FIFO and RR are better than CFS; for the other three benchmarks, CFS is better than FIFO and RR. In contrast, FLSCHED shows better performance for all benchmarks, except for *is*. Especially, four benchmarks (i.e., *cg*, *mg*, *sp*, and *ua*) show significantly better performance up to 1.73 times. For the analysis of scheduler behavior, we ran the *perf* profiling tool while running the benchmarks. We found that spinlock contention in the schedulers could become a major scalability bottleneck. As Table 2 shows, for the four benchmarks, which FLSCHED shows significant higher performance, CFS, FIFO, and RR spend significantly longer time for spinlock contention in schedulers (i.e., around 8-15%) than FLSCHED (i.e., around 3-5%). This shows that the increased sequential portion caused by lock contention in schedulers can significantly deteriorate the performance and scalability of applications.

To see how the number of context switching affects performance, we ran hackbench [11], which is a scheduler benchmark. We set the configuration of hackbench to use threads for parallelism and pipe for communication among threads. Figure 2 shows the execution time and the number of context switching on the Xeon Phi. It is clear that the number of context switching becomes a dominant factor of performance especially in communication-intensive applications such as hackbench. In particular, CFS shows the worst performance contrasting to FIFO, RR, and FLSCHED with the largest number of context switching. In Xeon Phi, the overhead of context switching is much higher than a Xeon processor due to its $2\times$ larger register set and $1.2\times$ slower memory bandwidth per core [9].

The above two cases show that all three representative schedulers fail to scale on manycore processors: all three have significant lock contentions, and CFS incurs too frequent context switching, which is expensive for manycore accelerators. This is the right moment to revisit the design and policy of schedulers, as manycore processors and accelerators are getting rapidly popular, and many applications have started being stuck by these problems.

3 FLSCHED: FEATHER-LIKE SCHEDULER

In this section, we describe the overview and major design principles of FLSCHED, and present its design in detail. To aim a novel scheduler on manycore processors like Xeon Phi, FLSCHED focuses on lockless implementation, decision mechanism for context switch for rescheduling and preemption, and efficient implementation.

3.1 Overview and design

From the version 2.6.23, the mainline Linux kernel adopted CFS scheduler as a default scheduler [16]. It is undisputed that CFS is very useful and one of the commonly-used schedulers, but CFS can break the performance of applications and still has chances to perform better as discussed in §2. From the case study on Xeon Phi, we derived the lessons as below.

Lock contention. It is well known that locking mechanism is essential for accessing shared data and, in the meanwhile, can break the scalability of system [17]. Unfortunately, with a lot of cores and as the number of cores increased, the performance breaking effect of a lock in scheduler increases with an exponential scale.

Context switches. Another lesson is that CFS incurs additional or too frequent context switches due to its design goal. From our code analysis of the latest CFS scheduler, CFS performs context switch for responsiveness, fairness, and load balancing. It is reasonable because CFS was designed for desktop and server machines from its birth. However, keeping responsiveness and fairness between users and tasks are less required for co-processors like Xeon Phi, and in that sense we need to reduce context switches.

Lock types	CORE	CFS	FIFO/RR	FL
raw_spin_lock	16	1	12	-
raw_spin_lock_irq/irqsave	13	5	2	-
rcu_read_lock	14	5	1	-
spin_lock	-	-	-	-
spin_lock_irq/irqsave	12	-	-	-
read_lock	3	-	-	-
read_lock_irq/irqsave	1	-	-	-
mutex_lock	6	-	-	-
Total	**65**	**11**	**15**	**0**

Table 1: Number of locks in the scheduler codes of the Linux kernel for Xeon Phi. FLSCHED is implemented without locks in itself.

Efficiency. CFS updates scheduling information as frequently as possible because CFS tries to strictly keep the fairness between tasks, and it is backed up by lots of task running states. With the small number of cores, it is meaningful to schedule tasks in a fine-grain manner for better fairness. However, if we have a lot of cores, making scheduling decision faster is more important rather than sophisticated decision determined by a lot of information updates because we can use many cores for each task.

Limitations and trade-offs. Although the lockless design and implementation of FLSCHED contributed to the performance enhancement, we unfortunately failed to reduce the locks in the scheduler core of the Linux kernel because the modification of scheduler core takes effect to all of other schedulers. Also, by reducing and delaying the context switches, FLSCHED can lose responsibility. However, FLSCHED achieves more throughput which is crucial for accelerators like Xeon Phi.

3.2 Locklessness

The performance degradation due to lock holding in scheduler has been pointed out for long time. Fortunately, Linux removed a global lock mechanism and distributed a centralized runqueue lock to per-runqueue locks from 2.6 kernel [20]. However, modern OS schedulers and CFS, which is state of the art, still embed locks.

Table 1 shows the number of locks in the scheduler codes of the Linux kernel for Xeon Phi. The core scheduler code (*sched.c*) includes the highest number of locks, CFS (*sched_fair.c*) has 11 locks, and FIFO/RR (*sched_rt.c*) has 15 locks, in total. Most of these locks are used for runqueue management, runtime statistics updates, load balancing mechanism, and scheduler features. On the other hand, FLSCHED is implemented without locks in itself by restructuring and optimizing of mechanisms.

Comparing to RR which has the largest number of locks, two locks are for the runtime statistics, five locks are to balance the load of cores, and eight locks are used for bandwidth control mechanism. First, we removed the runtime statistics updates of scheduler to reduce these locks. It is possible because these are not critical for FLSCHED to make scheduling decisions on Xeon Phi. The Linux scheduler adopts two types of load balance triggering mechanisms. One is triggered by a periodic timer, and the other is triggered by scheduler events such as *fork/exec* system calls and task wake-up. Five locks to keep balance the load of cores are used for the periodic load balance mechanism. FLSCHED is implemented not to use the periodic load balance to avoid the five locks. CPU bandwidth control is to support hard CPU bandwidth limits. A scheduler that is work-conserving by nature does not limit the maximum amount of CPU time. However, there are computing environments that need the maximum CPU time. Especially, cloud computing environment in which users pay per use should guarantee the maximum CPU bandwidth [26]. This feature is less required for co-processor like Xeon Phi, and therefore FLSCHED removes it with the last eight locks. The lock elimination of FLSCHED largely contributes to performance improvement as shown §4.1. Note that FLSCHED itself has no locks, but the scheduler core part has locks.

3.3 Less context switches

Reschedule. The main scheduler function checks a bit flag, *NEED_RESCHED* bit, to decide whether to execute a context switch or not. FLSCHED delays all settings of the reschedule flag to avoid context switches as many as possible. This approach has the possibility of decreasing response time and fairness. However, computation throughput is more important than responsiveness and fairness since Xeon Phi will be mostly used for high performance computing. Especially, the context switch overhead of Xeon Phi is much larger than that of general-purpose processors because its core contains more register sets for vector processing [9].

Preemption. Although the reasons of preemption are various case by case, most of preemption is incurred by priority. For most of OS, the priority mechanism is essential to differentiate the runtime or the order to get CPU. However, it is less required for Xeon Phi to compute data using massive cores because most of tasks will work using the same priority. When preemption is needed for any reason, FLSCHED does not immediately perform preemption. Instead, FLSCHED moves the location of tasks in runqueues and performs normal task switches in later term.

3.4 Faster and efficient scheduling decision

To make scheduler faster and more efficient, one of our approaches is to minimize scheduling information updates. In the case of CFS, the *update_curr_fair* function, whose major role is to renew the runtime statistics of current task, takes very short time but it is called huge number of times with a spinlock as shown in Table 3. It can be non-negligible overhead in manycore processors. FLSCHED works based on a given time slice with Round-Robin. The amount of time slice for a task is the only managed scheduling information for FLSCHED and managed inside of the *task_tick* function only.

The *enqueue_task/dequeue_task* functions take role of move a task from/to the runqueue of a CPU core. As the number of context switch increases, these function calls are also increased. FLSCHED simplified the total depth and path of these functions. Consequentially, the average execution time of these functions are less than 24% compared to those of CFS, as shown in Table 3.

FLSCHED does not provide three scheduling features that are supported in CFS: control groups, group scheduling, and autogroup scheduling. All these are not necessary features for Xeon Phi co-processor, and therefore FLSCHED does not implement them for more efficiency.

4 EVALUATION

This section describes the evaluation results and analysis of FLSCHED. The evaluations of FLSCHED are performed to address the following questions:

- **Performance impact on applications**: How much do the locklessness and the efficiency of FLSCHED affect on the performance of real-world applications? We ran HPC applications on Xeon Phi, and the results are presented at §4.1.
- **Design choices**: What does make FLSCHED get better performance? Using a micro-benchmark, we measured how much each design choice contributes to performance improvement. The results are shown at §4.2 and §4.3.
- **Efficiency analysis**: How much efficient is FLSCHED? We show the efficiency of FLSCHED by comparing with CFS scheduler at function execution level. The results are summarized at §4.3.

We performed all of the experiments on our 18-core machine (1-socket, 18 cores per socket, 2 threads per core, Intel Xeon E5-2699) equipped with 64GB memory. This machine is equipped with a Xeon Phi processor, 31S1P, that has 57 physical cores (4 threads per core) and 8 GB internal memory. We used the Linux kernel version 4.1.0-rc8 for host machine and 2.6.38 for Xeon Phi. The Linux kernel 2.6.38 is the latest kernel version that Intel officially releases and supports as of 06/2017, and the version of Intel manycore platform software stack (MPSS) to control Xeon Phi is 3.5.1. We ran all evaluations 10 times and reported their average. The error bars on each graph represent standard deviations of the 10 executions.

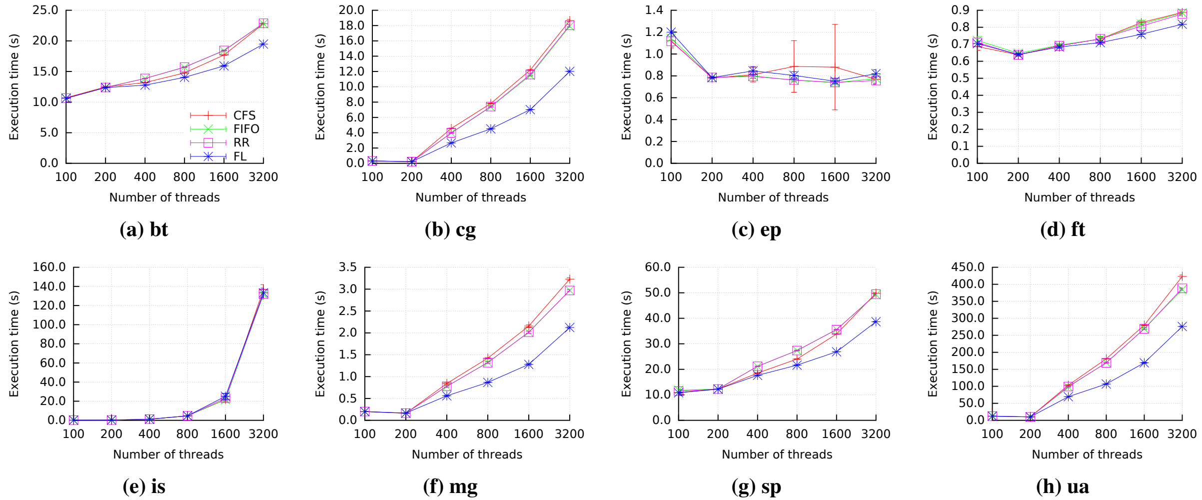

Figure 3: Performance comparison of NAS Parallel Benchmark (NPB) for the problem size class A with different schedulers.

4.1 Performance impact on applications

To show the performance impact on applications, we used the NAS Parallel Benchmark (NPB) [1]. NPB is a set of programs to evaluate the performance of parallel computing. We used the NPB version 3.3.1 which has ten evaluation benchmarks. Some of them cannot be run on Xeon Phi due to insufficient memory. As shown in Figure 3, we ran the programs with increasing number of OpenMP threads and measured the execution time of them. All of the applications show better performance with FLSCHED except the several cases of *ep* and *is*. Especially, FLSCHED shows much better performance than CFS, FIFO, and RR for *cg*, *mg*, *sp*, and *ua*. These four applications perform user-level spinning with calling *yield* system call within the OpenMP library waiting for a certain condition. As the scheduler invocation increases, the efficiency of scheduler largely affects to the overall execution time. In these cases, the major cause of performance gap among the designs of FLSCHED is the lockless design. As shown in Table 2, the *do_raw_spin_lock* function ratio in FLSCHED while executing the application is much lower than those of others. The small amount of lock contention in FLSCHED is due to the locks in the core code of the Linux scheduler (*core.c*). This evaluation result shows the small amount of lock elimination results in the large performance improvement of applications especially on manycore processors. Also, the lockless design of FLSCHED is highly effective in such cases.

NPB program	CFS (%)	FIFO (%)	RR (%)	FLSCHED (%)
bt	7.29	8.53	8.60	3.05
cg	**10.73**	**13.61**	**12.93**	**4.11**
ep	0.97	0.89	0.91	1.10
ft	5.34	5.25	5.57	4.04
is	0.21	0.17	0.18	0.12
mg	**6.84**	**7.30**	**7.15**	**2.85**
sp	**8.23**	**9.95**	**9.98**	**3.58**
ua	**14.63**	**15.79**	**15.46**	**5.96**

Table 2: Execution time of spinlock (*do_raw_spin_lock*) while executing NPB with 1,600 threads. We used the *perf* tool and collected system-wide information on Xeon Phi. The benchmark programs that FLSCHED shows much better performance are marked in bold.

4.2 Micro-benchmarks analysis

To deeply understand behavior of schedulers, we used hackbench [11] which is well known for scheduler performance evaluation. It creates task groups, and each group sends and receives short messages each other. Tasks can be created either of *fork* or *pthread*, and message passing can be via either of *socket* or *pipe*. We evaluated all combinations of them increasing parallelism as shown in Figure 4.

As the number of groups increases, the execution time of all schedulers increases in a linear scale, and the execution time of pipe message passing is much better than that of socket message passing due to socket overhead. In short, FLSCHED takes much shorter time than CFS and shows similar performance comparing with FIFO and RR. Especially, the performance improvement of FLSCHED is much higher in pipe message passing compared to CFS, since the scheduling overhead takes larger part of the total execution time with the smaller overhead of pipe than that of socket. In the case of hackbench running in a thread/socket mode, the improvement of FLSCHED is not large as much as the case of hackbench process with socket. However, FLSCHED is 3.12 times faster than CFS at best with pipe message passing.

Scheduler functions	CFS		FLSCHED		Normalized ratio
	Count	Average time (ns)	Count	Average time (ns)	(Average time)
check_preempt	42,184,784	5,058	3,202	917	0.18
dequeue_task	42,476,857	**19,008**	10,646	**3,636**	**0.19**
enqueue_task	42,479,016	**17,314**	10,792	**4,169**	**0.24**
pick_next_task	66,951,729	**5,261**	5,532,392	**1,937**	**0.37**
pre_schedule	-	-	10,646	718	-
put_prev_task	66,503,232	6,185	10,647	1,138	0.18
select_task_rq	42,426,871	**10,837**	8,031	**2,549**	**0.24**
set_cpus_allowed	-	-	1	2,997	-
task_tick	906,640	13,131	112	1,042	0.08
task_waking	42,418,867	2,290	10,792	1	0.00
update_curr	342,354,453	**2**	-	**-**	**0.00**

Table 3: Execution count and time of major scheduler functions while running hackbench on Xeon Phi. We ran hackbench with 200 groups of process and pipe communication. The total execution time was 28.037 seconds for CFS, and 11.102 seconds for FLSCHED, respectively. The important decision scheduler functions are marked in bold.

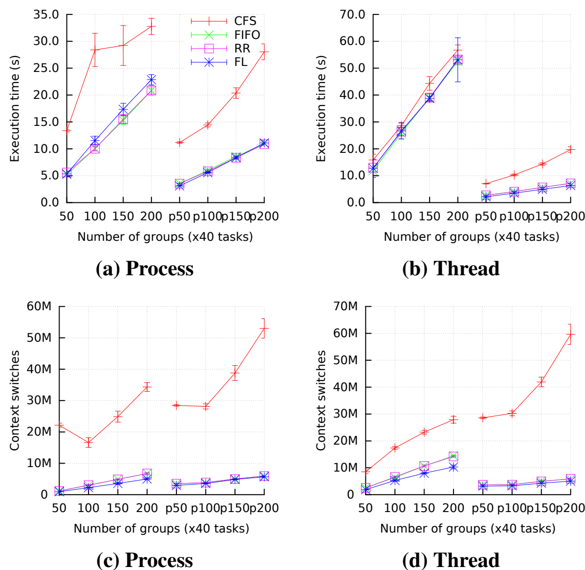

Figure 4: Execution time and number of context switches in hackbench. We tested the process/thread modes with socket/pipe message passing. A group uses 40 tasks, and therefore the number of group 200 means concurrent 8,000 tasks. The pXXX denotes pipe mechanism while the other denotes socket mechanism.

As performance increases in FLSCHED, the number of context switching decreases. In the case of p200, the context switches of CFS is 11 times more than that of FLSCHED. The context switch on Xeon Phi is more harmful than that on Xeon processor as described in §3.3. This results confirm that the design choice of FLSCHED to avoid context switches as many as possible is highly effective. In addition, the lockless design of FLSCHED has significant impact on the performance of hackbench. Note that the spinlock execution time in CFS takes 14.14% while that in FLSCHED is 8.13%.

4.3 Performance breakdown

To see what make the enhancement of FLSCHED in detail, we performed a function level analysis on CFS and FLSCHED. Table 3 shows the function call count and the average execution time of CFS and FLSCHED. We used the *ftrace* tool to collect the data while running hackbench process benchmark with 200 groups. During their executions, the call count and average execution time of each major scheduler function were collected. The "-" mark denotes that the function is not called. Each function call count is the sum of calls from each core and from multiple callers.

In Table 3, we can identify the average execution times of major scheduler functions in FLSCHED are much shorter than those of CFS. Especially, the *update_curr* function (*update_curr_fair* in CFS), which is called more than 342M times with 2ns, is completely not used in FLSCHED. Although the total execution time of *update_curr* is 684ms (2.4% of application execution time) with simple math, the effect of *update_curr* is exponentially amplified because it is protected by a spinlock, which increases the sequential portion of an entire system.

Moreover, the important scheduling decision functions (e.g. *enqueue_task*, *dequeue_task*, *pick_next_task*, and *select_task_rq*) take much shorter time. The average execution times of them are reduced by 63-81% from those of CFS. These ftrace evaluation results confirm that the design and implementation of FLSCHED, whose efficient decision and removal of unnecessary features to make it lightweight, is highly effective.

The function call counts of FLSCHED are also much lower than those of CFS. Although the execution time difference is 2.5 times, the call counts of functions are much higher. Especially, the important scheduling decision functions of CFS are 12-5,000 times more than those of FLSCHED. The *pre_schedule* and the *set_cpus_allowed* are added for FLSCHED,

but their performance overhead is negligible for the overall performance.

5 RELATED WORK

OS scheduler. The current state-of-the-art CFS Linux scheduler was introduced from the Linux kernel 2.6.23. Its main idea is that processes should be given a fair amount of processor by using the concept of virtual runtime [13]. However, the problem of CFS was addressed several times. Kravetz [17] showed the lock contention of the Linux runqueue and presented multiple runqueues. Lozi et al. [21] built a novel tool which could show scheduler activities and provided several bug fixes for the Linux kernel scheduler by using their tool. They are different from FLSCHED in that they tried to improve several faults and shortcomings of CFS, but our approach is to propose a novel scheduler for manycore processors like Xeon Phi.

Scheduler for Xeon Phi. Cadambi et al. [9] proposed a middleware for Xeon Phi, COSMIC, which performs a fair scheduling of multiple co-processors. Coviello et al. [10] studied a cluster scheduler for Xeon Phi compute clusters. As an extension of COSMIC, they implemented their sharing-aware algorithm onto CONDOR [25] and COSMIC. They are cluster level job schedulers while FLSCHED is an OS level scheduler for Xeon Phi itself. Jha et al. [15] analyzed the performance impact of thread affinity and schedulers which are supported by OpenMP for their hash join implementation on Xeon Phi. It is also a job allocation level scheduler, not an OS scheduler for Xeon Phi like FLSCHED.

6 CONCLUSION

We performed a comprehensive analysis on the Linux schedulers including CFS, the state-of-the-art scheduler of the Linux kernel, using a manycore processor. We observed performance degradation of the Linux schedulers and found the two major causes: spinlock contention in scheduler and too many context switches.

From the lessons learned, we propose a novel scheduler, FLSCHED, which is designed for lockless implementation and aims for less context switches. FLSCHED accomplished much faster and more efficient scheduling decision. FLSCHED shows better performance than CFS up to 1.73× for HPC applications and 3.12× for micro-benchmarks on Xeon Phi.

7 ACKNOWLEDGMENT

This work was supported by the ICT R&D program of MSIP/IITP (B0101-16-0644, The research project on High Performance and Scalable Manycore Operating System) and the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2017-2015-0-00378) supervised by the IITP(Institute for Information & communications Technology Promotion)

REFERENCES

[1] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications* 5, 3 (1991), 63–73.

[2] Jeff Barr. 2016. AWS Blog: X1 Instances for EC2 - Ready for Your Memory-Intensive Workloads. (2016). https://aws.amazon.com/blogs/aws/x1-instances-for-ec2-ready-for-your-memory-intensive-workloads/.

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 29–44.

[4] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. 2010. A case for NUMA-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 557–558.

[5] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, et al. 2008. Corey: An Operating System for Many Cores.. In *OSDI*, Vol. 8. 43–57.

[6] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. 2010. An Analysis of Linux Scalability to Many Cores.. In *OSDI*, Vol. 10. 86–93.

[7] Ray Bryant and John Hawkes. 2003. Linux scalability for large NUMA systems. In *Linux Symposium*. 76.

[8] bsdlazyfpu last visit: 09/09/2016. *NetBSD Documentation: How lazy FPU context switch works*. http://www.netbsd.org/docs/kernel/lazyfpu.html.

[9] Srihari Cadambi, Giuseppe Coviello, Cheng-Hong Li, Rajat Phull, Kunal Rao, Murugan Sankaradass, and Srimat Chakradhar. 2013. COSMIC: middleware for high performance and reliable multiprocessing on xeon phi coprocessors. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. ACM, 215–226.

[10] Giuseppe Coviello, Srihari Cadambi, and Srimat Chakradhar. 2014. A Coprocessor Sharing-Aware Scheduler for Xeon Phi-Based Compute Clusters. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 337–346.

[11] hackbench last visit: 09/09/2016. *Hackbench*. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c.

[12] Ashif S Harji, Peter A Buhr, and Tim Brecht. 2011. Our troubles with Linux and why you should care. In *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM, 2.

[13] IBM. last visit: 09/09/2016. *Inside the Linux 2.6 Completely Fair Scheduler*. https://www.ibm.com/developerworks/library/l-completely-fair-scheduler.

[14] Intel. last visit: 09/09/2016. *Intel Xeon Processor E7 Family*. http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family.html.

[15] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. 2015. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *Proceedings of the VLDB Endowment* 8, 6 (2015), 642–653.

[16] kernelcfs last visit: 09/09/2016. *CFS scheduler*. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[17] Mike Kravetz, Hubertus Franke, Shailabh Nagar, and Rajan Ravindran. 2001. Enhancing Linux scheduler scalability. In *Proceedings of the Ottawa Linux Symposium, Ottawa, CA*.

[18] Sanjeev Kumar. 2014. Efficiency at Scale. In *First International Workshop on Rack-scale Computing*.

[19] Daniel Kusswurm. 2014. Advanced Vector Extensions (AVX). In *Modern X86 Assembly Language Programming*. Springer, 327–349.

[20] Robert Love. 2005. *Linux Kernel Development (Novell Press)*. Novell Press.

[21] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM.

[22] Timothy Prickett Morgan. 2016. Intel Knights Landing Yields Big Bang For The Buck Jump. (2016). http://www.nextplatform.com/2016/06/20/intel-knights-landing-yields-big-bang-buck-jump/.

[23] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. 2008. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*. 27.

[24] Agam Shah. 2016. Intel's secretive Knights Mill mega-chip will challenge GPUs for AI domination. (2016). http://www.pcworld.com/article/3108799/components-processors/intels-knights-mill-mega-chip-to-take-on-gpus-in-ai.html.

[25] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. 2001. Condor: a distributed job scheduler. In *Beowulf cluster computing with Linux*. MIT press, 307–350.

[26] Paul Turner, Bharata B Rao, and Nikhil Rao. 2010. CPU bandwidth control for CFS. In *Linux Symposium 2010*. 245–254.

[27] Fenghua Yu. 2014. Ever Growing CPU States: Context Switch with Less Memory and Better Performance. In *LinuxCon North America*. Linux Foundation. http://events.linuxfoundation.org/sites/events/files/slides/LinuxCon_NA_2014.pdf.