Lock-Contention-Aware Scheduler: A Scalable and Energy-Efficient Method for Addressing Scalability Collapse on Multicore Systems

YAN CUI, YINGXIN WANG, YU CHEN, and YUANCHUN SHI, Tsinghua University

In response to the increasing ubiquity of multicore processors, there has been widespread development of multithreaded applications that strive to realize their full potential. Unfortunately, lock contention within operating systems can limit the scalability of multicore systems so severely that an increase in the number of cores can actually lead to reduced performance (i.e., scalability collapse).

Existing efforts of solving scalability collapse mainly focus on making critical sections of kernel code fine-grained or designing new synchronization primitives. However, these methods have disadvantages in scalability or energy efficiency. In this article, we observe that the percentage of lock-waiting time over the total execution time for a lock intensive task has a significant correlation with the occurrence of scalability collapse. Based on this observation, a lock-contention-aware scheduler is proposed. Specifically, each task in the scheduler monitors its percentage of lock waiting time continuously. If the percentage exceeds a predefined threshold, this task is considered as lock intensive and migrated to a Special Set of Cores (i.e., SSC). In this way, the number of concurrently running lock-intensive tasks is limited to the number of cores in the SSC, and therefore, the degree of lock contention is controlled. A central challenge of using this scheme is how many cores should be allocated in the SSC to handle lock-intensive tasks. In our scheduler, the optimal number of cores is determined online by the model-driven search.

The proposed scheduler is implemented in the recent Linux kernel and evaluated using micro- and macrobenchmarks on AMD and Intel 32-core systems. Experimental results suggest that our proposal is able to remove scalability collapse completely and sustains the maximal throughput of the spin-lock-based system for most applications. Furthermore, the percentage of lock-waiting time can be reduced by up to 84%. When compared with scalability collapse reduction methods such as requester-based locking scheme and sleeping-based synchronization primitives, our scheme exhibits significant advantages in scalability, power consumption, and energy efficiency.

Categories and Subject Descriptors: D.4.1 [Operating Systems]: Process Management; D.4.8 [Operating Systems]: Performance

General Terms: Measurement, Design, Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Scalability collapse, operating systems, multicore systems, lock-contention-aware scheduler

ACM Reference Format:

Cui, Y., Wang, Y., Chen, Y., and Shi, Y. 2013. Lock-Contention-Aware scheduler: A scalable and energy efficient method for addressing scalability collapse on multicore systems. *ACM Trans. Architect. Code Optim.* 9, 4, Article 44 (January 2013), 25 pages.

DOI = 10.1145/2400682.2400703 <http://doi.acm.org/10.1145/2400682.2400703>

This work is supported by the National Science Foundation of China, under grant 61170050.

Authors' addresses: Y. Cui (corresponding author), Y. Wang, Y. Chen, and Y. Shi, Computer Science and Technology Department, Tsinghua University, Beijing, China; email: ccuiyyan@gmail.com.

Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART44 \$15.00

DOI 10.1145/2400682.2400703 <http://doi.acm.org/10.1145/2400682.2400703>

1. INTRODUCTION

In order to address the issues of power and performance with single-core processors, major chip manufacturers including Intel, AMD, and IBM have switched to multicores. Today, multicore processors have been in widespread use in almost all computing environments. Trends seen from Intel's 80-core [Vangal et al 2008] and Tiler's 100-core chip [Tiler 2012] suggest that thousands of cores will be integrated into a single chip in just ten years [Agarwal and Levy 2007; Boyd-Wickizer et al. 2008; Wentzlaff and Agarwal 2009].

To release the performance potential of multicores, applications are often designed to be fully parallel so that tasks scheduled on different cores can be executed simultaneously. However, lock contention in operating systems can significantly limit application scalability and may result in scalability collapse, an anomaly in which increasing the number of cores leads to reduced speedup [Appavoo et al. 2007; Boyd-Wickizer et al. 2008; Cui et al. 2011]. Figure 1 demonstrates the scalability of parallel postmark, a macrobenchmark designed for evaluating file server performance, on a Linux-based AMD 32-core platform. As shown in the figure, an increase in the number of threads initially results in an increase in throughput; after the thread number continues to grow, however, speedup experiences a sharp decrease. Our performance data collected from profiling tools indicate that scalability collapse is caused by two kinds of ticket spin lock contention in Linux.

For kernel-lock-intensive applications such as parallel postmark, two key factors dominate their scalability. One is the sequential execution introduced by critical sections, the other is the overhead caused by the lock implementation. Assume the application runs on Linux system, which implies ticket spin lock is used for synchronization. When only a few contended tasks exist, very few tasks wait to enter critical sections of kernel code and the overhead of acquiring locks is negligible compared with the lengths of critical sections. At this stage, the lock can be acquired without incurring any cache coherence overhead. Thus, the effect of lock contention contributes less to the overall application throughput. When the number of application tasks increases, the speedup can be improved, but the probability of lock contention also increases and each lock operation introduces more and more cache coherence overhead. At this time, the lengths of critical sections become relatively short compared with the execution time overhead of acquiring locks. If the number of contended tasks becomes so large that a particular critical section of kernel code becomes the main throughput bottleneck, the time used to wait for the lock protecting the critical section increases linearly with the number of lock requesters and becomes significantly larger than the execution time of the critical section. (This means the critical section *becomes* rather short.) At this phase, each lock acquisition and releasing can complete only after suffering many cache coherence misses. This is the way in which scalability collapse occurs.

Scalability collapse occurs when locks are heavily contended. The conventional wisdom is that real-life applications, such as web servers and OLTP (OnLine Transaction Processing) applications, rarely suffer scalability collapse because of lock contention. However, with the advent of many-core systems, we believe that this is a view that needs to change. Although the probability of lock contention was small on small-scale systems, when switched to large-scale systems, the contention intensity could be significantly higher and the overhead of acquiring a lock could be significantly larger than the length of the critical section. This fact and several recent studies [Wentzlaff and Agarwal 2009; Boyd-Wickizer et al. 2010; Cui et al. 2011] contribute to our belief that scalability collapse caused by lock contention has the potential to become an increasingly serious problem on multicores.

Using fine-grained locks is a traditional way of avoiding scalability collapse. However, this method has two main disadvantages. First, it cannot solve scalability collapse

Fig. 1. Overall throughput and average percentage of lock-waiting time of all threads in parallel postmark on a Linux-based AMD 32-core machine. The details of benchmark description, the experimental setup, and the hardware architecture are introduced in Section 4.

fundamentally because collapse is only deferred, and it occurs again on systems with more cores. Second, it is becoming increasingly difficult and time consuming to adopt this method in the many-core era because making critical sections of kernel code fine-grained should keep pace with the number of increased cores [Wentzlaff and Agarwal 2009], but critical sections in today's commodity operating systems (e.g., Linux and Solaris) are already rather short.

Another way is to modify the mechanism of synchronization primitive to avoid scalability collapse. For example, mutex lock has tasks wait for a lock by sleeping instead of constantly polling. However, the context switch overhead of this method can cause unsatisfactory scalability. The use of an adaptive lock is a natural method to overcome the major disadvantage of the mutex lock because of its ability to make trade-offs adaptively between polling and sleeping. However, the decision of when to poll or when to sleep is usually decided heuristically [adaptive spinning mutexes 2009; McDougall and Mauro 2006; McKusick and Neville-Neil 2004] and achieving the full potential of adaptive locks is hindered. As an example, the most popular policy of determining whether to spin or sleep at the time of waiting for an adaptive lock is based on the status of the lock holder. When a task (A) requests for a lock that is currently held by another task (B), task A does not sleep if task B is in the running status, expecting the lock would be released soon. Only when task B is sleeping, task A goes to sleep. As we can imagine, when the number of tasks is smaller than the number of cores, the adaptive lock behaves like a spin lock because the lock holder is always running, and scalability collapse may still happen when an adaptive lock is heavily contended, so the scalability collapse cannot be avoided by the use of adaptive locks. Adopting scalable synchronization primitives (e.g., MCS [Mellor-Crummey and Scott 1991] or CLH [Magnussen et al. 1994] locks) can also be used to address scalability collapse. However, these kinds of locks make all lock requesters constantly poll a local flag while waiting, and therefore prevents other tasks from running, which reduces energy efficiency and resource utilization [Hammarlund et al. 2008]. Besides, these kinds of locks exploit complex atomic instructions, which are not scalable. Last but not least, the scheduling method proposed in this article can be used together with scalable locks to provide excellent scalability and energy efficiency.

To avoid these disadvantages in previous methods, a novel lock-contention-aware scheduler is proposed in this article. Our proposal is motivated by an observation that

the percentage of lock-waiting time for a lock-intensive task starts to increase significantly at the same point when scalability collapse occurs. Based on this observation, our scheduler keeps track of the percentage on a per-task basis. If a task's percentage exceeds a predefined threshold, the task is migrated to a Special Set of Cores (i.e., SSC). In this way, the number of concurrently running tasks is limited, and therefore, the degree of lock contention is controlled. However, a challenge is how to online decide the number of cores in the SSC. Our solution is to develop a throughput model for lock-intensive tasks and the optimal number of cores is determined by the model-driven search. Besides, the optimal value can adapt the change of the locking behavior by monitoring several key parameters of our scheduler (i.e., the number of cores in the SSC) and firing a timer at a configurable period.

Our lock-contention-aware scheduler is implemented in the Linux kernel 2.6.29.4 and evaluated on AMD 32-core and Intel 32-core systems. Experimental results using micro- and macrobenchmarks indicate that our scheduler can remove scalability collapse and sustain the maximal throughput of the spin-lock-based system for most applications. Furthermore, the percentage of lock-waiting time can be reduced by up to 84%. When compared with collapse reduction methods such as requester-based locking scheme and sleeping-based synchronization primitives, our method offers significant improvements in scalability, power consumption, and energy efficiency.

The central contribution of this article is the design, implementation, and evaluation of the lock-contention-aware scheduler. The rest of this article is organized as follows. Section 2 analyzes the scalability collapse phenomenon from the perspective of a task's percentage of lock-waiting time. Section 3 presents the design and implementation of our lock-contention-aware scheduler. Section 4 describes the benchmarks, experimental setups, and results. Section 5 relates this research to previous work and Section 6 summarizes our conclusions.

2. SCALABILITY COLLAPSE ANALYSIS

This section correlates the scalability collapse phenomenon with the percentage of lock-waiting time for a lock-intensive task. The curve in Figure 1 (labeled as *percentage*) shows the total percentage of waiting time for two hottest spin locks (i.e., `tmpfs` statistics lock and file descriptor table lock) contended in parallel postmark. The data displayed in the figure is the average of all threads where the number of cores was varied to see its effect on lock contention on an AMD 32-core system.

As indicated in this curve, the average percentage of lock-waiting time increases slowly from 3.81% to 21.77% before the occurrence of scalability collapse. However, after the number of cores is larger than 16, scalability collapse happens and the percentage grows rapidly to 92.24%. From the experimental results, we conclude that scalability collapses because of lock contention in the kernel along with the percentage of lock-waiting time disproportionately increase after collapse occurs. Besides, if an application suffers from scalability collapse, each thread becomes so busy with waiting for a spin lock that almost no time is left for the useful work.

3. LOCK-CONTENTION-AWARE SCHEDULER

3.1. Overview

The observation made in Section 2 motivates us to design a lock-contention-aware scheduler. Overall, our scheduler is made up of four parts.

—*Scalability Collapse Detection.* Each task in the scheduler monitors its percentage of lock-waiting time continuously after it is created. If the percentage is larger than a predefined threshold, the task is identified as lock intensive and migrated to an SSC.

- Core Allocation in the SSC.* All lock-intensive tasks are running on the SSC. This part determines how many cores should be used in the SSC to achieve the maximal normalized throughput for lock-intensive tasks¹.
- Load Balancing Separation.* To overcome the system-wide load balancing of the default scheduler, our scheduler balances lock-intensive and nonlock-intensive tasks separately.
- Locking Behavior Adaption.* The change of the locking behavior is detected by monitoring several key parameters of our scheduler (e.g., the number of cores in the SSC) and firing a timer in a configurable period. If the locking behavior changes, the number of cores in the SSC is redetermined.

3.2. Scalability Collapse Detection

The lock-contention-aware scheduler identifies a task as lock intensive and migrates it to an SSC when the task spends a considerable amount of time on waiting for locks. To qualify whether a task is lock intensive, we calculated its percentage of lock-waiting time during each time slice (i.e., the interval of two consecutive context switches). The percentage of lock-waiting time is formulated as $\sum_j W_{i,j}/D_i$, where $W_{i,j}$ represents the waiting time for j^{th} lock acquisition in time slice i and D_i represents the duration of time slice i . In order to calculate $\sum_j W_{i,j}$ in the formula, we provide the function `sched_lock()` to replace `spin_lock()`, which is the default kernel function of acquiring a spin lock. The pseudocode of `sched_lock()` is presented in Algorithm 1. As can be seen, `sched_lock()` is a wrapper of `spin_lock()`. Besides, it adds the migration logic (`migrate_to_special_cores()`) and the fine-grained measurement of the lock acquisition interval by the use of the time stamp counter (TSC). For D_i , it is also calculated by reading the TSC at the start and end of each time slice. Notice that, although TSC reading is performed frequently to measure the time interval of each lock acquisition and the length of each time slice, it is implemented by issuing the `rdtsc` instruction, which is especially lightweight. In Section 4.5, we will demonstrate the overhead is negligible.

ALGORITHM 1: `sched_lock()`

Input: The spin lock pointer `lock`

- ```

1 /*Current task is migrated based on the accumulated lock-waiting time*/
 migrate_to_special_cores(lock);
2 curr→start = read_tsc();
3 spin_lock(lock);
4 curr→acc_lock_time += read_tsc() - curr→start;
```
- 

At the end of each time slice, the scheduler will check whether the current task has been considered as lock intensive (lines 1 to 9 in Algorithm 3) in the `schedule()` function, which is responsible for task scheduling in Linux. If not, the scheduler calculates the percentage of lock-waiting time for this task. If the percentage is larger than a predefined threshold  $T$ , an entry `is_mig` in the `task_struct` data structure is set to `QUALIFY_TO_MIGRATE`, indicating the task has been considered as lock intensive. To determine the threshold, an empirical value of 10% is used. In Section 4.5, we will see that this value is effective in avoiding scalability collapse.

The practical migration to a SSC occurs when a task attempts to acquire a spin lock. Algorithm 2 presents the pseudocode of `migrate_to_special_cores()`, which is invoked in

---

<sup>1</sup>Assume  $K(n)$  represents the measured throughput when running all lock-intensive tasks on  $n$  cores. Thus, the normalized throughput for lock-intensive tasks is defined as  $\frac{K(n)}{K(1)}$ .

Fig. 2. The process of core allocation and release in the SSC.

*sched\_lock()*. As can be seen, if *is\_mig* of the current task has been set to be *QUALIFY\_TO\_MIGRATE*, it is updated to *HAVE\_MIGRATED* and the task will be migrated by invoking *sched\_migrate\_task()*. The target core of the migration is selected to make the load of all cores in the SSC as balanced as possible. Notice that the current task can also be migrated when *is\_mig* is *HAVE\_MIGRATED*. This is necessary because the optimal number of cores in the SSC is determined online by the model-driven search. In the next section, we will cover the details.

---

**ALGORITHM 2:** *migrate\_to\_special\_cores()*

---

**Input:** The spin lock pointer *lock*

```

1 local_core_bound = special_core_bound;
2 if (curr→is_mig==QUALIFY_TO_MIGRATE) or (curr→is_mig==HAVE_MIGRATED and
 curr→special_core_bound ≠ local_core_bound) then
3 if curr→is_mig==QUALIFY_TO_MIGRATE then
4 inc_lock_intensive_tasks();
5 curr→is_mig=HAVE_MIGRATED;
6 end
7 /*get a core whose id is chosen from 0 to local_core_bound*/;
8 target_core = get_target_core(local_core_bound);
9 /*migrate curr task to target_core*/;
10 if this_core ≠ target_core then
11 sched_migrate_task(curr, target_core);
12 end
13 curr→special_core_bound=local_core_bound;
14 end

```

---

### 3.3. Core Allocation in the SSC

All tasks which are identified as lock intensive are executed on an SSC. In our lock-contention-aware scheduler, cores with consecutive id are allocated for the SSC. Figure 2 demonstrates how the allocation and release of cores in the SSC are performed from an initial state (two cores in the SSC). A global variable *special\_core\_bound* is used to distinguish the range of cores in the SSC and the remaining cores. Note that our core allocation method sufficiently considers the characteristic of our multicore platforms, where cores with consecutive id are integrated in the same chip. For example, cores with id  $[4 \times i, 4 \times (i + 1))$ ,  $0 \leq i \leq 7$  locate on the same chip in our AMD platform (see Section 4.1). Therefore, this core allocation method ensures that the least number of nodes will be used on NUMA systems, which avoids many expensive node-to-node cache line transfer. Besides, this core allocation method also ensures that the sharing degree of last-level caches is maximal, which makes the communication between cores in the SSC fast. For systems where cores with consecutive id do not

exist in the same chip, the benefits of our method can also be acquired by simply adding an extra core id mapping layer in software.

The key issue of our scheduler is how many cores should be allocated in the SSC to handle lock-intensive tasks. If only a few cores are allocated, the probability of lock contention is small and the shared data in critical sections is bounced infrequently. However, the parallelism of the application is limited. In contrast, if too many cores are allocated, tasks can execute in parallel but scalability collapse may still exist. Actually, the optimal number of cores in the SSC (i.e.,  $n^*$ ) is achieved by solving an optimization problem, which is expressed as

$$n^* = \arg \max_{1 \leq n \leq \min\{M, N\}} T(n), \quad (1)$$

where  $T(n)$  represents the normalized throughput of all lock-intensive tasks when  $n$  cores are allocated in the SSC,  $M$  represents the number of lock-intensive tasks, and  $N$  represents the total number of cores in system. As indicated in Eq. (1), our objective is to maximize the normalized throughput of all lock-intensive tasks.

Assume  $p_i(n)$  represents the percentage of lock-waiting time for core  $i$  when  $n$  cores are allocated in the SSC and  $\bar{p}(n)$  represents the average percentage of lock-waiting time of all cores in the SSC. Then,  $T(n)$  is approximated by the following equation.

$$T(n) = \sum_{i=1}^n (1 - p_i(n)) = n \cdot \left( 1 - \frac{\sum_{i=1}^n p_i(n)}{n} \right) = n \cdot (1 - \bar{p}(n)) \quad (2)$$

Here  $1 - p_i(n)$  represents the percentage of *useful work* on core  $i$ . In this equation, only lock contention bottleneck is considered. Although the model is simple, as we will see in the evaluation part, it is effective for our purpose.

One possible way of solving this optimization problem is to search the optimal  $n$  in a brute-force manner. The initialized value of  $n$  is set to be one and each time  $n$  is increased by one. Once  $n$  is updated,  $\bar{p}(n)$  and  $T(n)$  is recalculated. The process lasts until the maximal  $T(n)$  is found. However, this method is infeasible in practice because the time complexity of the search process is high. Searching for the optimal  $T(n)$  has the complexity of  $O(N)$ , where  $N$  is the number of cores in the system. However,  $N$  will increase exponentially in the future and hence it is unbearable that the complexity keeps the same growth rate.

To reduce the overhead, one heuristic rule is proposed to accelerate the search. If  $T(n)$  is larger than the last calculation,  $n$  doubles and the search continues. Or else, the search stops and  $n$  is set to be the value in the last calculation. It is obvious that the proposed rule reduces the time complexity (reduced to  $O(\log N)$ ) by sacrificing accuracy. However, this method has an acceptable effect on the accuracy and works well for a wide range of workloads, as will be shown in Section 4.

To calculate  $T(n)$ ,  $\bar{p}(n)$  is a key parameter. Our scheduler estimates  $\bar{p}(n)$  by the use of voting. Specifically, the state of the voting system is represented by two global variables, *voting\_locking* and *voting\_slice*. The former records the total time spent in waiting for locks, while *voting\_slice* accumulates the total time slice for all voting tasks. Using these two variables,  $\bar{p}(n)$  is calculated as  $\frac{\text{voting\_locking}}{\text{voting\_slice}}$ .

The pseudocode of core allocation and release in the SSC is presented in Algorithm 3. At the end of each time slice, each lock-intensive task is given the chance to vote (line from 10 to 19 in Algorithm 3). Each voting task tries to acquire the lock protecting voting variables. If the lock is held successfully, *voting\_locking* and *voting\_slice* accumulate the total time in waiting for locks (i.e., *acc\_lock\_time*) and the time slice length (i.e., *slice\_len*) of the current task, respectively (line 16). Notice that the lock protecting the voting variables will not become a bottleneck because the lock is always acquired

**ALGORITHM 3:** increased code at the end and start of each time slice in schedule()

---

```

Input:
1 /*runs at the end of each time slice*/;
2 curr→slice_end=read.tsc();
3 curr→slice_len=curr→slice_end - curr→slice_begin;
4 if curr→is_mig==0 then
5 if curr→acc_lock_time>T×curr→slice_len then
6 | curr→is_mig=QUALIFY_TO_MIGRATE;
7 | curr→special_core_bound=special_core_bound;
8 end
9 else
10 if curr→is_mig==QUALIFY_TO_MIGRATE or curr→is_mig==HAVE_MIGRATED
11 then
12 | /*the second condition indicates special_core_bound has been modified by another
13 | | task*/;
14 | can_vote=(this_cpu<special_core_bound and
15 | | curr→record_bound==special_core_bound);
16 | if can_vote then
17 | if spin_trylock(voting_lock) then
18 | if curr→record_bound==special_core_bound then
19 | | VOTE(1, n_lock_intensive_tasks, curr→acc_lock_time, curr→slice_len);
20 | end
21 | spin_unlock(voting_lock);
22 end
23 | is_sufficient=voting_slice > special_core_bound × 256000;
24 | if is_sufficient and spin_trylock(voting_lock) then
25 | | determine_new_bound();
26 | | spin_unlock(voting_lock);
27 end
28 end
29 end
30 /*time out logic*/;
31 if timer_fires() and spin_trylock(voting_lock) then
32 | if timer_fires() then
33 | | special_core_bound=1;
34 | | CLEAR_VOTING();
35 | | climbing=1;
36 | | remig_next=jiffies + remig_timeout_interval;
37 end
38 | spin_unlock(voting_lock);
39 end
40 curr→acc_lock_time=0;
41
42 /*runs at the begin of each time slice*/;
43 curr→slice_begin=read.tsc();
44 curr→record_bound=special_core_bound;

```

---

using *spin\_trylock()*, whose semantic ensures that a lock is not waited for when it cannot be acquired immediately. (Actually, the function returns in this case.) To verify this conclusion experimentally, we use a microbenchmark named single counter (see Section 4.2), where each process increases a spin lock protected global counter in a tight loop. We use two methods to acquire this lock. One is *spin\_trylock()* and the other is *spin\_lock()*. The measurement for these two methods indicates that the cost of acquiring



the lock using *spin\_trylock()* is only one-fifteenth of the cost using *spin\_lock()* when this lock is heavily contended.

After voting, the current task judges whether a sufficient number of cores have voted by calculating the total voting slice per special core. If the value is large enough, the current task tries to acquire the spin lock protecting the voting variables again. If the lock is acquired successfully, the task is responsible for driving the search of optimal number of cores in the SSC. Notice that this method prevents multiple tasks from making different decisions simultaneously because only one core can acquire the lock at any time. Furthermore, after a core drives the search, the voting variables are cleared or shrunk (see Section 4), which avoids performing the search too frequently.

All the search logic is handled by the function *determine\_new\_bound()*, which is invoked in Algorithm 3 and is detailed in Algorithm 4. As can be seen in Algorithm 4, the normalized throughput is first calculated (line from 2 to 4). Then, it is compared with the normalized throughput calculated in the last step to decide whether to double the number of cores in the SSC or step back. However, the measurement error can make the throughput inaccurate. To overcome this situation, we change the number of cores in the SSC only if the throughput is larger or smaller than the last calculation two consecutive times. The logic is described using a state machine (line from 5 to 43). Once the number of cores in the SSC is updated successfully, all voting variables are reset to zero to start a new round of voting, or else, they are divided by two to reflect the effect of time (line from 48 to 50). Note that if our scheduler decides to reduce the number of cores in the SSC, the search process will stop until the locking behavior changes (line from 33 to 37).

A question is how lock-intensive tasks are aware of the latest number of cores in the SSC. Recall that a lock-intensive task may still migrate to a core in the SSC even if it has been migrated once (see Algorithm 2). It happens when the current number of cores in the SSC is different from the task's recorded value after the last migration. By the use of this mechanism, each lock-intensive task can notice the latest number of cores in the SSC when attempting to acquire a spin lock and all lock-intensive tasks will run on cores with id from 0 to *special\_core\_bound*−1.

### 3.4. Load Balancing Separation

Our scheduler manages all lock-intensive tasks on cores in the SSC. However, the load may not be balanced system-wide. Thus, our scheduler provides separate load balancing among cores inside and outside the SSC to avoid the effect of default global load balancing. Specifically, in the separate load balancing, the core running the balancing functions (*find\_busiest\_group()*, *find\_busiest\_queue()* and *run\_rebalance\_domains()*) first checks the *special\_core\_bound*. If the id of this core is smaller than this value, all balancing is only performed on cores with id from 0 to *special\_core\_bound*−1, or else, the load balancing is performed on remaining cores. Although the load balancing is separated, the methodology of balancing in each set is the same with the default system.

### 3.5. Locking Behavior Adaptation

The locking behavior of an application can change because tasks in the system are created and terminated dynamically, or the spin locks frequently acquired by lock-intensive tasks may change. Once the locking behavior changes, our lock-contention-aware scheduler needs to redetermine the optimal number of cores in the SSC for lock-intensive tasks. Our scheduler exploits the number of cores in the SSC and the normalized throughput to detect the change of the locking behavior. The pseudocode is presented in Algorithm 4 (lines from 44 to 46). As we can see, if the number of cores in the SSC is not equal to the value of last round or the normalized throughput becomes

**ALGORITHM 4:** determine\_new\_bound()

---

```

Input:
1 moving=0;
2 useful_perc = 1 - voting_locking/voting_slice;
3 work_cores = n_lock_intensive_tasks > special_core_bound ? special_core_bound : n_lock_intensive_tasks;
4 thro = useful_perc × work_cores; //calculate the normalized throughput;
5 // climbing is initilized to be 1;
6 if climbing then
7 if thro > thro_last_step then
8 if climbing < 0 then
9 climbing=1;
10 else
11 if climbing == 2 then
12 new_bound = 2 × work_cores;
13 if new_bound > max_special_cores then
14 new_bound = max_special_cores;
15 end
16 if new_bound > n_lock_intensive_tasks then
17 new_bound = n_lock_intensive_tasks;
18 end
19 if new_bound > work_cores then
20 special_core_bound = new_bound; //bound update;
21 RECORD_LAST_VOTING(n_lock_intensive_tasks, work_cores, thro);
22 CLEAR_VOTING();
23 moving=1;
24 end
25 climbing= 1;
26 else
27 climbing++;
28 end
29 end
30 else
31 if climbing > 0 then
32 climbing= -1;
33 else
34 if climbing == -2 then
35 special_core_bound = work_cores_last_step;
36 CLEAR_VOTING();
37 moving=1;
38 climbing=0;
39 else
40 climbing- -;
41 end
42 end
43 end
44 else
45 if work_cores ≠ work_cores_last or throughput > $\alpha \times \text{throughput_last_step}$ or
 throughput < $\beta \times \text{throughput_last_step}$ then
46 remig_next=jiffies; //fire timer to recalculate the number of cores in the SSC;
47 end
48 end
49 if moving == 0 then
50 SHRINK_VOTING(1, 2);
51 else
52 remig_next=jiffies + remig_timeout_interval;
53 end

```

---

overly larger or smaller than the last calculation, the timer, which is responsible for invoking the core reallocation in the SSC, is fired. The timeout handler is presented in Algorithm 3 (lines from 26 to 35), where we reinitialize the *special\_core\_bound* and the state of the voting system. Notice that another way to fire the timer is the core allocation in the SSC stays unchanged for *remig\_timeout\_interval*. The interval is empirically set to be 30 seconds in our system.

### 3.6. Discussion

**3.6.1. Fairness.** Our lock-contention-aware scheduler limits the execution of all lock-intensive tasks on the SSC. However, the algorithms of selecting a task from all the waiting tasks and balancing tasks among cores in the SSC are exactly the same with the default Linux. Thus, for lock-intensive tasks, the fairness is not hurt although not all CPUs are used.

**3.6.2. Target Workloads.** Our scheduler targets for homogeneous kernel-lock-intensive workloads, where each task performs similar operations and the same set of spin locks are contended for by all the tasks at any time. Actually, many real-life applications can be modeled as homogeneous kernel-lock-intensive workloads, such as file servers and mail servers.

## 4. EVALUATION

### 4.1. Multicore Platform and Operating System

The effectiveness of our lock-contention-aware scheduler is verified on Linux 2.6.29.4-based AMD 32-core and Intel 32-core NUMA systems. Each will be introduced in turn.

For the AMD 32-core system, there are eight Opteron 8347HE chips and each chip has four cores. Each core owns a private L1 data cache, L1 instruction cache, and L2 cache. The size of each L1 cache is 64K bytes, while the size of each L2 cache is 512K bytes. Four cores on the same chip share the same L3 cache. The size of the L3 cache is 2M bytes. Intra-chip cores and separate chips are connected by the internal crossbar switch and HyperTransport, respectively. The 32G memory is partitioned into 8 banks, where each bank connects to one of the 8 chips. The task migration overhead on this system is also measured. The latency is 9.3 microseconds between two cores on the same chip, 9.7 microseconds between two cores one hop away, 10.2 microseconds two hops away, and 11.6 microseconds three hops away.

The Intel 32-core system is equipped with four Xeon X7560 processors and each processor owns eight cores. Hyperthreading technology is applied in the system and each core can support two hardware threads. Each core has private 32K bytes instruction and data caches and private 256K bytes L2 cache. Eight cores on the same chip share one 24M bytes L3 cache. Total memory of the system is partitioned into four banks, each connecting to one processor. During benchmarking, hyperthreading capability is disabled to ease the analysis of scalability. Note that our experiments are mainly performed on the AMD 32-core system. Unless specified otherwise, the reported experimental results are collected from this system.

### 4.2. Benchmarks and Running Methodology

We use micro- and macrobenchmarks to verify the effectiveness of our lock-contention-aware scheduler. Single counter [Sridharan et al. 2006], mmapbench [Cui et al. 2011], and sockbench [Cui et al. 2011] are microbenchmarks, while parallel postmark [Cui et al. 2010], kernbench [Con Kolivas 2006], parallel find [Roszbach et al. 2007], and parallel grep are macrobenchmarks. We select these benchmarks because their most significant scalability bottleneck is spin lock contention and they cover a wide range

of contention intensity. All these benchmarks are homogeneous workloads, where each task performs similar work.

All microbenchmarks are implemented as multiprocess programs and synchronized using the same framework. The single counter benchmark has each process increase the same counter protected by a spin lock in the kernel space; mmapbench has each process map the same continuous 500MBytes with the MAP\_SHARED flag from a file, touch each page by reading the first byte, and destroy the mapping; sockbench has each process create a socket and then close the result. Two parameters can be tuned in all these microbenchmarks. One is the number of processes that is currently running and the other is the number of operations each process performs in a test.

Parallel postmark is a multithreaded benchmark, which has the capability of simulating file servers providing email and netnews services. Each thread in parallel postmark executes transactions repeatedly on an independent set of files (between 0.5 and 10K bytes in size) and each transaction is made up of two steps: (1) creating or deleting a file (2) reading or appending a file. Files, file I/O operations (e.g., create and read), and file sizes are chosen from a uniform distribution. To measure the scalability of the Linux kernel instead of I/O, files in the workload are created in tmpfs. Furthermore, file operations without I/O buffering are used to avoid the buffering effect of glibc. Performance is measured by the transaction throughput of all threads. In this benchmark, the initial number of files for each thread is tunable and we create two workloads by changing this number. One workload uses 500 initial files for each thread and the other uses 10000 files.

Kernbench measures the performance of a particular system by compiling the kernel using multiple concurrent processes. In our experiments, we use kernbench to compile Linux kernel 2.6.29.4. For each test, we only change the number of concurrent processes with the fast run option enabled and half load run and optimal load run options disabled. Like parallel postmark, the kernel source code used by kernbench is also created in tmpfs to eliminate the I/O effect.

Parallel find creates multiple processes and each process in the benchmark calls GNU find repeatedly to search files on an independent copy of Linux kernel 2.6.29.4 for a nonexistent file name. All copies of the Linux kernel source code are under the same file system and have the same path depth of five. The ratio of dataset size to memory size is 0.3068 ( $\frac{32 \times 306.8MB}{32GB} = 0.3068$ ). There are two tunable parameters in this benchmark. One is to control how many processes are created and the other is to control how many times each process performs the “find” operation. Hardware caches are warmed up before the actual test.

Parallel grep generates multiple processes and each process calls GNU grep to search a text string that cannot be found in a set of small files. The total size of the file set is 4G. All small files have the same path depth of six. There are also two tunable parameters for parallel grep. The purpose of these parameters are the same with parallel find. When running this test, all files are stored in the tmpfs.

Among these seven benchmarks, parallel postmark, kernbench, and parallel grep use tmpfs to store the working directories and files. Tmpfs instead of a real file system is used for two reasons. First, we have replaced tmpfs with a real file system. However, the largest bottleneck becomes I/O instead of lock contention, but our focus in this article is to solve the lock contention bottleneck. Second, in industry, I/O is not necessarily the actual bottleneck for file-operation-intensive applications as RAID is usually used to improve the IOPS (Input/output Operations Per Second). Thus, tmpfs is used in our test to simulate RAID because we do not have such a system.

As we always keep the same number of processes or threads as that of tested cores, and each process or thread generates the same amount of workload, the system

Table I. Hot Kernel Locks and Corresponding Hot Functions Invoking *spin\_lock()* for All Benchmarks

| Benchmarks               | Hot Kernel Locks                         | Hot Invoking Functions                                                                                                         |
|--------------------------|------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| <i>mmapbench</i>         | mmapings lock ( <i>i_mmap_lock</i> )     | <code>unlink_file_vma()</code> <code>vma_link()</code>                                                                         |
| <i>sockbench</i>         | dentry cache lock ( <i>dcache_lock</i> ) | <code>d_alloc()</code> <code>_atomic_dec_and_lock()</code><br><code>d_instantiate()</code> <code>d_rehash()</code>             |
|                          | inode lock ( <i>inode_lock</i> )         | <code>new_inode()</code> <code>_mark_inode_dirty()</code><br><code>generic_sync_sb_inodes()</code>                             |
| <i>parallel postmark</i> | file descriptor table lock               | <code>sys_close()</code> <code>fd_install()</code><br><code>get_unused_fd_flags()</code>                                       |
|                          | files lock ( <i>files_lock</i> )         | <code>file_kill()</code> <code>file_move()</code>                                                                              |
| <i>kernbench</i>         | files lock ( <i>files_lock</i> )         | <code>file_move()</code> <code>file_kill()</code>                                                                              |
|                          | dentry cache lock ( <i>dcache_lock</i> ) | <code>d_alloc()</code> <code>_atomic_dec_and_lock()</code><br><code>d_path()</code> <code>dcache_readdir()</code>              |
| <i>parallel find</i>     | files lock ( <i>files_lock</i> )         | <code>file_kill()</code> <code>file_move()</code>                                                                              |
|                          | dentry cache lock ( <i>dcache_lock</i> ) | <code>_atomic_dec_and_lock()</code>                                                                                            |
| <i>parallel grep</i>     | files lock ( <i>files_lock</i> )         | <code>file_move()</code> <code>file_kill()</code>                                                                              |
|                          | per dentry lock ( <i>d_lock</i> )        | <code>_d_lookup()</code> <code>dnotify_parent()</code> <code>dput()</code><br><code>inotify_dentry_parent_queue_event()</code> |

Hot locks are the same for two configurations of parallel postmark.

scalability for a benchmark can be characterized as the change of throughput with the number of cores. In a fully scalable system, the overall throughput will increase at the same rate as the number of cores.

### 4.3. Hot Kernel Spin Locks

`/proc/lock_stat` in Linux is enabled to identify which kernel locks are heavily contended during benchmarking [Pepper 2007]. The hot kernel spin locks for each benchmark are presented in Table I. The results for single counter are omitted because the spin lock in the benchmark is defined by us.

### 4.4. Measurement Methodology

To measure the performance of our proposed scheduler, we replace the hot invoked points of each hot spin lock reported by `/proc/lock_stat` with our lock wrapper (i.e., `sched.lock()`). Note that, although not all spin lock invoked points in the kernel are replaced, it is enough to verify our proposed idea because the lock contention is dominated by contention of hot locks in the hot code paths. Furthermore, our experiments show a significant scalability improvement when replacing the hot code paths of hot locks, which we believe offers ample evidence to verify the potential of our scheduling method on an even broader scale.

In Section 4.9, we will compare our lock-contention-aware scheduler with other scalability reduction methods, including mutexs, adaptive locks, and requester-based locks. Our measurement methodology for these methods is to replace the hot spin locks and all invoking points of each hot spin lock. Different from our scheduling scheme, to replace hot code paths of a spin lock is not feasible because different synchronization primitives cannot be used simultaneously.

### 4.5. Scalability Improvements

Figure 3 presents the throughput of our lock-contention-aware scheduler on seven workloads. The experimental results in the default Linux are also presented in the same figure for comparison. As indicated in this figure, for four of six benchmarks

Fig. 3. Throughput of different scalability reduction schemes when varying the number of cores. For post-mark, the number of initial files is also presented. The throughput of single counter is in log scale.

suffering from scalability collapse (i.e., mmapbench, sockbench, parallel postmark, and parallel grep), the proposed lock contention-aware scheduler captures the start of collapse accurately and sustains the maximal speedup. For one benchmark which does not suffer scalability collapse (i.e., kernbench), the lock-contention-aware scheduler performs almost the same as the default scheduler. As will be seen in the next section, almost no task is identified as lock intensive during the execution of kernbench. Thus, the task migration overhead is not introduced and the unique source of performance degradation in the lock-contention-aware scheduler comes from continuously monitoring each task's percentage of lock-waiting time. According to the experimental results of kernbench, the overhead of reading TSC is negligible.

However, two exceptions are noticed. First, for single counter, although the lock-contention-aware scheduler completely avoids scalability collapse when using more than one core, the maximal speedup of the default scheduler is not sustained. Second, for parallel find, the throughput of our lock-contention-aware scheduler starts to decrease slightly when using more than 15 cores. Although the throughput does not decrease so significantly as in the default scheduler, it suggests there are some nonscalable factors when running parallel find on our scheduler. These exceptional phenomena

Fig. 4. Number of allocated cores and lock-intensive tasks in our lock-contention-aware scheduler.

will be explained clearly by the investigation into the number of lock-intensive tasks and allocated cores in the SSC.

#### 4.6. Number of Lock-Intensive Tasks and Allocated Cores in the SSC

Figure 4 plots the number of identified lock-intensive tasks and the number of allocated cores in the SSC against the number of worker tasks for all benchmarks in our lock-contention-aware scheduler. For each number of worker tasks, we collect the number of lock-intensive tasks and allocated cores in the SSC three times. All these results are integrated into one figure. Recall that we determine the optimal number of cores in the SSC by model-driven search and the search is also invoked when the locking behavior change is detected. Thus, the number of lock-intensive tasks and allocated cores in the SSC can vary with time. In Figure 4, the results are collected when the search stabilizes.

The experimental results of mmapbench, sockbench, parallel postmark, and parallel grep suggest that our lock-contention-aware scheduler identifies all tasks as lock, intensive when more than one task is started. This result is reasonable because all of our benchmarks is homogeneous, which means all worker tasks have the same locking behavior. We can also see that the number of allocated cores in the SSC is always

optimal when combining with the scalability results of the default Linux system in Figure 3. Furthermore, the number of lock intensive tasks and allocated cores in the SSC for these four benchmarks are almost the same across different runs, indicating our scheduler is able to find the optimal number of cores very consistently. For kernbench, almost no tasks are identified as lock intensive because lock contention is rather lightweight. Thus, the number of lock-intensive tasks and allocated cores remain the initial value.

The inefficiency of our lock-contention-aware scheduler for single counter and parallel find can also be uncovered by the results in Figure 4. For single counter, our scheduler allocates two cores in the SSC to handle lock-intensive tasks with a very high probability. Thus, the maximal throughput cannot be sustained (see Figure 3(a)). The number of cores in the SSC is not optimal because of the measurement error in our core allocation algorithm. One possible explanation for this error is that the voting process starts once lock intensive tasks are identified. However, it may start before all lock intensive tasks are identified and migrated to one of the cores in the SSC. If this happens, the *voting\_locking* variable could be significantly larger than the ideal value and the number of allocated cores in the SSC cannot sustain the maximal throughput of a workload. Note that we have adopted several methods (e.g., update the number of cores in the SSC when the number of votes is large enough) in the practical system to bypass this effect, but this problem can still affect the optimal number of cores in the SSC for single counter, where all tasks seriously contend for a single spin lock.

For parallel find, the number of identified lock-intensive tasks fluctuates dramatically across the three runs when starting more than seven worker tasks. This happens because each created process in parallel find is excessively short-lived. Because each task completes in short time, processes are created and terminated frequently. Recall that our scheduler identifies a task as lock intensive after the *is\_mig* entry in the *task\_struct* data structure is set to be `QUALITIFY_TO_MIGRATE` (lines from 3 and 4 in Algorithm 2). If a task is too short-lived, a task may exit before it is marked as lock intensive. Thus, different number of lock-intensive tasks can be reported in different runs. Remember that our lock-contention-aware scheduler redetermines the number of cores in the SSC to adapt the change of the locking behavior. In parallel find, the frequent process creation and termination make the behavior change from time to time. Furthermore, the frequency of behavior change increases with the number of worker tasks. As a result, the throughput decreases gradually with the number of worker tasks. Also notice that when collapse occurs, the number of allocated cores in the SSC for parallel find is eight instead of the optimal value ten because our scheduler doubles the number of cores in the SSC after each round of voting.

Figure 4 also reveals an inefficiency of our lock-contention-aware scheduler. When the collapse just occurs, our scheduler tends to allocate more cores than the optimal (e.g., parallel postmark using 17 or 18 working tasks). This indicates that there is still room for improving the performance of load balancing in our scheduler. Fortunately, as shown in Figure 3, the inefficiency has modest effect to the overall scalability.

#### 4.7. CPU Utilization and Lock Contention Reduction

Figure 5 presents the average execution time breakdown of all busy cores for each benchmark on the default system and system with our lock-contention-aware scheduler when varying the number of tasks in each application. From this figure, we can see the execution time breakdown on the default system and system with our scheduler are similar: The CPU utilization of each busy core in single counter, mmapbench, sockbench, parallel postmark, and parallel grep are 100% and most of time is used to execute in the kernel mode. For kernbench, most of the execution time is cost in the user mode and each busy core starts to exhibit a particular percentage of idle when using

Fig. 5. The CPU utilization of each benchmark on the default system and system with our scheduler.

Fig. 6. The percentage of lock-waiting time in our lock-contention-aware and the default schedulers.

more than 4 cores. For parallel find, there is the largest percentage of idle among these benchmarks because each process is rather short-lived and many cores are reported to be busy.

The lock contention reduction is measured by comparing the percentage of the lock-waiting time in our proposed scheduler and the default scheduler. Figure 6 presents the experimental results when varying the number of tasks for one microbenchmark (mmapbench) and two macrobenchmarks (parallel postmark and kernbench). Results for other benchmarks are omitted for the sake of brevity. As expected, once scalability collapse occurs, the lock-contention-aware scheduler reduces the percentage of lock-waiting time greatly because all lock-intensive tasks are limited on the SSC to avoid collapse. Furthermore, the reduction becomes larger as the number of worker tasks increases. As a result, when 32 worker tasks are exploited, the percentage of the lock-waiting time is reduced by up to 84.0%.

#### 4.8. Computer Architecture Effects

One concern with our lock-contention-aware scheduling scheme is whether or not it is still effective on other system architectures. Actually, many architectural parameters may affect the effectiveness of our lock-contention-aware scheduler, such as cache size, cache coherence protocol, bandwidth of the memory controller, interconnection topology, etc. This section presents the scalability improvements of the lock-contention-aware scheduler on an Intel 32-core system with kernel version 2.6.29.4 to explore possible architectural effects.

Figure 7 presents results of three benchmarks (i.e., sockbench, parallel postmark, and parallel find). Results of other benchmarks are omitted. From Figure 7, one can see

Fig. 7. Throughputs of our lock-contention-aware and the default schedulers on an Intel 32-core system.

that the results on the Intel 32-core system are similar to those on the AMD 32-core system. The scalability collapse in sockbench and postmark are completely avoided after the adoption of the lock-contention-aware scheduler. Furthermore, the maximal throughput of the spin-lock-based system is sustained. For parallel find, the scalability is significantly improved in our proposed scheduler. However, the throughput decreases slightly with the number of worker tasks because the tasks in parallel find are rather short-lived. Thus, the overhead of searching the optimal number of cores in the SSC increases with the number of worker tasks.

The experimental results on the Intel system are similar to those on the AMD system, implying that the effectiveness of our lock-contention-aware scheduler is not sensitive to the changes of architectural parameters. This conclusion is expected because our scheduler does not depend on any specific hardware parameters although the changes of parameters can affect application performance.

#### 4.9. Comparing with Other Scalability Reduction Methods

**4.9.1. Comparison Targets.** Our goal in this article is to provide scalable performance and good energy efficiency for kernel-lock-intensive applications. To achieve this goal, there are two directions. First, spin locks are still used for synchronization, and other policies are used together with spin locks to overcome scalability collapse or ensure good energy efficiency. For example, this article demonstrates how to combine the scheduling technique and ticket spin lock protocol, while our previous proposal [Cui et al. 2012](requester-based locking) shows the combination of ticket spin lock and power-saving instructions. In this research direction, the policies used with spin locks are orthogonal to various spin lock mechanisms (e.g., scalable spin lock, ticket spin lock, etc). Second, spin locks are given up and we start to use block-based synchronization primitives. In this direction, power-saving policies are implied in the locking protocol. For example, the lock requesters using mutex locks will be put to sleep if the contended lock cannot be acquired immediately. Thus, cores will not held by blocked lock requesters and will not consume as much power as a busy core.

To offer a comparison to our lock-contention-aware scheduler, we select one solution (requester-based lock [Cui et al. 2012]) in the first research direction and two solutions (mutex locks and adaptive locks [adaptive spinning mutexes 2009]) in the second. Notice that we do not directly compare with classical scalable spin locks (e.g., MCS or CLH) because our scheduling technique is orthogonal to the scalable spin locks and these two techniques should be used together to provide a complete solution. We also do not compare with scalable locks with another power-saving policy (e.g., entering into power-saving state while waiting) because our current scheduler assumes the usage of ticket spin lock. Thus, the differences of spin lock implementations will invalidate

experimental results. In the Section 4.10, the applicability of our scheduler to scalable locks will be presented.

In these three selected solutions, requester-based locking decides the method of lock waiting (spinning or entering into the power-saving state) according to the current number of lock requesters. If the current number is estimated to be larger than zero, a newly arriving requester will enter into the power-saving state by the use of monitor and mwait instructions, or else, the requester will acquire the lock by spinning. For the mutex lock, it will put lock requesters to sleep if the contended lock cannot be acquired immediately. As for adaptive locks, the lock requesters will spin or sleep according to the running state of current lock holder.

*4.9.2. Scalability Comparison.* Figure 3 illustrates the throughput of different scalability collapse reduction methods as a function of the number of cores. In terms of microbenchmarking, our lock-contention-aware scheduler and the requester-based lock scale much better than the mutex lock and the adaptive lock for mmapbench and sockbench. This result is expected because the context switch overhead of sleeping-based synchronization primitives is rather large in addressing scalability collapse. When comparing our scheme with the requester-based lock, our method totally wins in these two cases, although the requester-based lock can also avoid scalability collapse completely.

As for single counter, our scheme performs better than the requester-based lock. However, these two schemes scale worse than the sleeping-based synchronization primitives. This happens because the context switch overhead cannot dominate the overall performance of sleeping-based synchronization primitives in single counter. Specifically, the mutex lock and the adaptive lock will try to acquire the lock before sleeping. In single counter, there is only one short critical section. Thus, the task just releasing the lock will tend to acquire the lock again. This mechanism makes the lock tend to be held by the same task without incurring the overhead of context switch. Actually, the better scalability of sleeping-based synchronization primitives is acquired by sacrificing the fairness.

In terms of macrobenchmarking, our method and requester-based lock scale much better than sleeping-based primitives in parallel postmark, parallel find, and parallel grep. When comparing our scheme with the requester-based lock, our scheme performs no worse than the requester-based lock in parallel postmark and parallel grep (especially for the 10000 initial files configuration) but worse in parallel find after 16 cores. The worse scalability in parallel find is due to frequent process creation and termination, which introduces relatively large scheduling overhead. For kernbench, our scheme, the requester-based lock, and the adaptive lock perform nearly the same with the original ticket-spin-lock-based system because lock contention in this benchmark is rather lightweight. One interesting phenomenon in kernbench is that scalability collapse occurs by the use of mutex, although the throughput in the original system keeps increasing with the number of cores. Recall that requesters of a mutex will sleep if this mutex cannot be acquired immediately. The context switch overhead incurred by the use of mutex is so large that scalability collapse occurs.

*4.9.3. Power Consumption Comparison.* Our lock-contention-aware scheduler uses an SSC to handle lock-intensive tasks. The remaining cores are idle and will not consume as much power as the cores in the SSC. This section compares the power consumption of different scalability reduction methods. To measure the overall power consumed during the execution of a workload, we adopt the 380801 power analyzer [380801 Power Analyzer 2012]. The power reported by the analyzer is relatively stable during the execution of all benchmarks. For each workload, the power is read three times and the mean value is reported. Notice that we measure the system power instead of the CPU

Fig. 8. Power consumption of different scalability reduction schemes when varying the number of cores.

power because our proposal also reduces the interconnection and memory activities, which cannot be reflected by the CPU power.

Figure 8 presents the power consumption of various scalability reduction methods when varying the number of cores. For single counter, mmapbench, sockbench, parallel postmark, parallel find, and parallel grep, the results are similar. The default system (with ticket spin lock) power consumption increases with the number of cores the most rapidly among all the solutions because each lock requester waits for a lock by constantly spinning, which is rather power hungry. For requester-based lock, the power consumption also increases with the number of cores, but slower. This is because most lock requesters will save power using monitor and mwait instructions while waiting. The lock-contention-aware scheduler, mutex lock, and adaptive mutex lock are three solutions that consume the least power. As we can see, the power consumption of these methods will stabilize with an increased number of cores. The experimental results also show that leaving processor idle is an efficient mechanism of saving power.

For kernbench, the power consumption of our lock-contention-aware scheduler, the requester-based lock, the mutex lock, and the adaptive lock are almost the same. This is expected because the lock contention in this benchmark is lightweight. The only exception is the mutex lock, whose power consumption starts to decrease when using

Fig. 9. Energy efficiency of different scalability reduction schemes when varying the number of cores.

more than 29 cores. Recall that the mutex lock can cause scalability collapse when using more than 29 cores (see Figure 3(e)). Once collapse occurs, a particular lock is heavily contended, and therefore, all lock requesters are put to sleep, which degrades the total power consumption.

**4.9.4. Energy-Efficiency Comparison.** In this section, we compare various collapse reduction methods in energy efficiency, which is quantified as completed work per joule (i.e.,  $\text{efficiency} = \frac{\text{throughput} \times \text{time}}{\text{power} \times \text{time}} = \frac{\text{throughput}}{\text{power}}$ ). Figure 9 presents the experimental results. Overall, the lock-contention-aware scheduler exhibits reasonable energy-efficiency improvements. Specifically, for microbenchmarks, the energy efficiencies of our scheduler are better than requester-based lock and sleeping-based synchronization primitives for mmapbench and sockbench. For single counter, our scheme is better than requester-based lock, but worse than sleeping-based synchronization primitives. Sleeping based primitives perform better in this case because fairness is not ensured.

For macrobenchmarks, our scheme and the requester-based locking scheme also exhibit excellent energy efficiency in parallel postmark, parallel find, and parallel grep, compared with mutex locks and adaptive locks. When comparing our scheduler with the requester-based lock, our method wins in parallel grep and two configurations

Fig. 10. Throughput, power, and energy efficiency of mmapbench and parallel find on default system, system with MCS lock, and system with lock-contention-aware scheduler when varying the number of cores.

of parallel postmark. For parallel find, the efficiency of our scheduler is worse than that of requester-based lock when using more than 18 cores due to frequent task migration.

#### 4.10. Applicability of Our Scheduler to Other Spin Locks

Our implementation of lock-contention-aware scheduler assumes the use of ticket spin lock. However, the proposed scheduling technique does not depend on and can be used together with any specific spin lock protocol. To demonstrate this, our lock-contention-aware scheduler is combined with one implementation of scalable spin locks (i.e., MCS lock) to improve scalability and energy efficiency.

MCS lock [Mellor-Crummey and Scott 1991] is a well-known lock protocol to avoid scalability collapse. When waiting for an MCS lock, each lock requester spins on a local flag instead of a global variable to minimize expensive cache line bouncing among cores. When a lock is released, only the lock requester with the longest waiting time will be notified. Figure 10(a) and (d) present the scalability improvements of one microbenchmark (i.e., mmapbench) and one macrobenchmark (i.e., parallel find) when replacing the ticket spin lock protocol in the default system with the MCS lock (labeled as “MCS”). As can be seen from these figures, using MCS locks can avoid scalability collapse completely for applications where kernel locks are not highly contended (e.g., parallel find), but collapse still exists for applications with serious lock contention (e.g., mmapbench). The throughput of mmapbench in the system with MCS lock can decrease slightly with the number of cores because MCS exploits complex atomic instructions in its implementation, which is not scalable. The overhead can be further enlarged in the microbenchmark test. For parallel find, although scalability collapse is completely avoided by the use of MCS lock, the power consumption increases rapidly with the number of cores (see Figure 10(e)).

The advantage of combining our lock-contention-aware scheduler with the MCS lock is clear: For applications in which scalability collapse can be totally avoided by the use of MCS lock, the combination provides better power consumption and energy efficiency;

for applications for which collapse cannot be totally avoided, the combination also provides better scalability. Our lock-contention-aware scheduler is modified to use the lock-waiting time of MCS lock. Throughput, power consumption, and energy efficiency of one microbenchmark and one macrobenchmark on a system with a modified scheduler (also labeled as “lock-contention-aware scheduler”) are presented in Figure 10. The results of the default system and system with MCS locks are also presented for comparison. As can be seen, combining MCS lock with our scheduler can achieve better scalability than using MCS lock alone for mmapbench. This is because MCS lock uses complex nonscalable atomic instructions and the overhead can be enlarged in highly contended applications such as mmapbench. For parallel find, combining our scheduler and the MCS lock generates similar scalability with using the MCS lock alone. When looking at the power consumption, combining our scheduler with the MCS lock is much better than the system with MCS lock and the default system. This happens because our scheduler limits contended tasks on an SSC and leaves some cores idle. As a result, less power is consumed. Because the throughput of our scheduler (combined with MCS lock) is no less than that of using MCS alone and the power consumption of our scheduler is much less, the energy efficiency of our scheduler becomes much better.

## 5. RELATED WORK

In this article we have demonstrated that scalability can collapse due to spin lock contention in operating systems. The root cause is the sequential execution of critical sections and the overhead of spin lock implementation. To solve the scalability collapse, existing efforts mainly focus on making critical sections fine-grained or designing new synchronization primitives. Although these techniques reduce the effect of scalability collapse, they have disadvantages in scalability or energy efficiency. Specifically, using fine-grained locks runs the risk of suffering scalability collapse again on systems with more cores. Besides, it is increasingly difficult and time consuming to make critical sections of kernel code fine-grained. Designing new synchronization primitives needs the support of complex atomic instructions or context switch, which are expensive. Thus, collapse cannot be avoided when the primitive is highly contended. Furthermore, the scalable locks (e.g., MCS [Mellor-Crummey and Scott 1991] and CLH locks [Magnussen et al. 1994]) make all lock requesters wait by constantly polling, which reduces the energy efficiency and resource efficiency [Hammarlund et al. 2008]. Orthogonal to these methods, our lock-contention-aware scheduler controls the number of concurrently running tasks, and therefore limits the degree of lock contention. Note that our method features excellent programmability, scalability, and energy efficiency. In terms of programmability, it is relatively easy to integrate our scheduler into commodity operating systems. Besides, once collapse is avoided by the use of our scheduler, it will not occur again no matter how many worker tasks are created. As for scalability, our scheduler is able to avoid collapse completely and sustains the maximal speedup of the spin-lock-based system for most applications. For energy efficiency, all lock-intensive tasks are handled on the SSC. The remaining cores are left idle, and therefore, they will not consume as much energy as the cores in the SSC.

Addressing scalability collapse by the use of scheduling has been investigated by F. Xian et al. [2008]. In their paper, they propose to exploit the contention-aware scheduler to reduce lock contention for multithreaded Java programs. The largest difference between their scheduler and ours is the target workloads. For their scheduler, each selected Java program is a heterogeneous workload and most efforts are put on clustering threads contending for different sets of locks. However, for our scheduler, it aims at homogeneous workloads. Thus, the focus is how to find the optimal number of cores in the SSC to handle lock-intensive tasks.



Another research related with ours is sharing-aware scheduling [Tam et al. 2007]. In the proposal, threads in the system are clustered based on the data sharing patterns to reduce total inter-chip latency. Although the sharing of lock data can also introduce large chip-to-chip latency, this method cannot improve the scalability of our target applications because it only works for heterogeneous workloads where threads can be classified into multiple sets based on the relationship of data sharing.

The early design of our lock-contention-aware scheduler is motivated by the Accelerated Critical Sections (ACS) [Suleman et al. 2009] and the *kernel core* abstract in Corey [Boyd-Wickizer et al. 2008]. ACS accelerates critical sections on asymmetric multicore systems by executing critical sections on high-performance cores. When a critical section is encountered, the computation is migrated between low- and high-performance cores and new instructions are added to perform the migration of critical sections. In Corey, the *kernel core* abstract allows applications to dedicate cores to manage system calls and the requests of hardware devices sent from other cores. Shared-memory IPC is exploited to migrate computation. We have implemented the lock-contention-aware scheduler using a similar idea. A few cores are reserved to execute critical sections. When encountering critical sections, computation is migrated to the reserved cores; after completing the execution of critical sections, computation is migrated back. However, experimental results suggest that the performance is especially bad because of migration overhead. The presented version of lock-contention-aware scheduler performs task migration based on throughput, which is more coarse-grained than migration based on the execution of critical sections.

Our lock-contention-aware scheduler belongs to nonconserving scheduling, which leaves some cores idle even when there are tasks waiting for the computing resources [Rosti et al. 1995; Smirni et al. 1995]. Besides scalability collapse caused by spin lock contention, the nonconserving scheduling can also be exploited for preventing collapse caused by other reasons. As a motivational example, Fedorova et al. implement a user-level scheduler to avoid collapse caused by cache contention on SMT processors [Fedorova and Smith 2006].

## 6. CONCLUDING REMARKS

This article introduces the lock-contention-aware scheduler to address scalability collapse caused by lock contention on multicores. Orthogonal to traditional collapse reduction methods, our solution exploits the SSC to execute lock-intensive tasks. Thus, the number of concurrently running lock-intensive tasks are limited and the degree of lock contention is controlled. One challenge of the proposed method is how to determine the optimal number of cores in the SSC as it could be different for different applications. To solve this challenge, our scheduler searches the optimal number of cores driven by a throughput model. Results on two 32-core platforms using micro- and macrobenchmarks suggest that the proposed scheduler completely avoids scalability collapse and sustains the maximal throughput of spin-lock based system. Furthermore, our scheduler shows significant advantages in scalability, power consumption, and energy efficiency compared with previous methods.

## REFERENCES

- 380801 POWER ANALYZER. 2012. <http://extech.com/instruments/product.asp?catid=14&prodid=205>.
- ADAPTIVE SPINNING MUTEXES. 2009. <http://lkml.org/lkml/2009/1/14/393>.
- AGARWAL, A. AND LEVY, M. 2007. The kill rule for multicore. In *Proceedings of the Design Automation Conference (DAC)*. ACM, 750–753.
- FEDOROVA, M. S. AND SMITH, M. D. 2006. A non-work-conserving operating system scheduler for smt processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*. 10–17.



- APPAVOO, J., SILVA, D. D., KRIEGER, O., AUSLANDER, M., OSTROWSKI, M., ET AL. 2007. Experience distributing objects in an smp os. *ACM Trans. Comput. Syst.* ACM, New York.
- BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KASHOEK, F., ET AL. 2008. Corey: An operating system for many cores. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.
- BOYD-WICKIZER, S., CLEMENTS, A., MAO, Y., PESTEREV, A., KAASHOEK, M. F., ET AL. 2010. An analysis of linux scalability to many cores. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.
- CON KOLIVAS. 2006. <http://ck.kolivas.org/kernbench/>.
- CUI, Y., CHEN, Y., SHI, Y., AND WU, Q. 2010. Parallel scalability comparison of commodity operating systems on large scale multi-cores. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 117–18.
- CUI, Y., WANG, Y., CHEN, Y., AND SHI, Y. 2011. Experience on comparison of operating systems scalability on the multi-core architecture. In *Proceedings of the IEEE International Conference on Cluster Computing*. 205–215.
- CUI, Y., WANG, Y., CHEN, Y., SHI, Y., HAN, W., ET AL. 2012. Reducing scalability collapse via requester-based locking on multi-core systems. In *Proceedings of the IEEE 20<sup>th</sup> International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 298–307.
- HAMMARLUND, P., CROSSLAND, J. B. AGGARWAL, A., AND KAUSHIK, S. D. 2008. Queued locks using monitor-memory wait. US Patents: US 20080022141A1.
- MAGNUSSEN, P., LANDIN, A., AND HAGERSTEN, E. 1994. Queue locks on cache coherent multi-processors. In *Proceedings of the 8<sup>th</sup> International Symposium on Parallel Processing (IPPS)*. 165–171.
- MCDUGALL, R. AND MAURO, J. 2006. *Solaris Internals Solaris 10 and OpenSolaris Kernel Architecture*, 2nd Ed.
- McKUSICK, M. K. AND NEVILLE-NEIL, G. V. 2004. *The Design and Implementation of the FreeBSD Operating System*.
- MELLOR-CRUMMEY, J. AND SCOTT, M. L. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* 9, 1, 21–65.
- PEPPER, T. 2007. Linux kernel lock profiling with lockstat. <http://dolavim.us/blog/archives/2007/11/linux-kernel-lo.html>.
- ROSSBACH, C. J., HOFMAN, O. S., PORTER, D. E., RAMADAN, H. E., ADITYA, B., ET AL. 2007. Txlinux: Using and managing hardware transactional memory in an operating system. In *Proceedings of the 21<sup>st</sup> ACM Symposium on Operating Systems Principles (SOSP)*. ACM, New York, 87–102.
- ROSTI, E., SMIRNI, E., SERAZZI, G., AND DOWDY, L. W. 1995. Analysis of non-work-conserving processor partition policies. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*. 165–181.
- SMIRNI, E., ROSTI, E., SERAZZI, G., DOWDY, L. W., AND SEVCIK, K. C. 1995. Performance gains from leaving idle processors in multiprocessor systems. In *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*. 203–210.
- SRIDHARAN, S., KECK, B., MURPHY, R., CH, S., AND KOGGE, P. 2006. Thread migration to improve synchronization performance. In *Proceedings of the Workshop on Operating System Interference in High Performance Applications*.
- SULEMAN, M. A., MUTLU, O., QURESHI, M. K., AND PATT, Y. N. 2009. Accelerating critical section execution with asymmetric multi-core architecture. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, 253–264.
- TAM, D., AZIMI, R., AND STUMM, M. 2007. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the EuroSys Conference*. ACM, New York, 47–58.
- TILERA, 2012. <http://www.tilera.com/products/processors/TILE-Gx.Family>.
- VANGAL, S., HOWARD, J., RUHL, G., DIGHE, S., WILSON, H., ET AL. 2008. An 80-tile sub-100-w teraflops processor—Wen in 65-nm cmos. *IEEE J. Solid-State-Circuits* 43, 1, 29–41.
- WENTZLAFF, D. AND AGARWAL, A. 2009. Factored operating systems(fos): The case for a scalable operating system for multicores. *ACM SIGOPS Oper. Syst. Rev.*
- XIAN, F., SRISA-AN, W., AND JIANG, H. 2008. Contention-aware scheduler: Unlocking execution parallelism in multithreaded java programs. In *Proceedings of the 23<sup>rd</sup> Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 163–180.

Received 2012; revised September 2012; accepted November 2012