Shuffling: A Framework for Lock Contention Aware Thread Scheduling for Multicore Multiprocessor Systems

Kishore Kumar Pusukuri Rajiv Gupta Laxmi N. Bhuyan
Department of Computer Science and Engineering
University of California, Riverside
Riverside, USA 92521
{kishore, gupta, bhuyan}@cs.ucr.edu

ABSTRACT

On a cache-coherent multicore multiprocessor system, the performance of a multithreaded application with high lock contention is very sensitive to the distribution of application threads across multiple processors (or Sockets). This is because the distribution of threads impacts the frequency of lock transfers between Sockets, which in turn impacts the frequency of last-level cache (LLC) misses that lie on the critical path of execution. Since the latency of a LLC miss is high, an increase of LLC misses on the critical path increases both lock acquisition latency and critical section processing time. However, thread schedulers for operating systems, such as Solaris and Linux, are oblivious of the lock contention among multiple threads belonging to an application and therefore fail to deliver high performance for multithreaded applications.

To alleviate the above problem, in this paper, we propose a scheduling framework called *Shuffling*, which migrates threads of a multithreaded program across Sockets so that threads seeking locks are more likely to find the locks on the same Socket. Shuffling reduces the time threads spend on acquiring locks and speeds up the execution of shared data accesses in the critical section, ultimately reducing the execution time of the application. We have implemented Shuffling on a 64-core Supermicro server running Oracle Solaris 11™ and evaluated it using a wide variety of 20 multithreaded programs with high lock contention. Our experiments show that Shuffling achieves up to 54% reduction in execution time and an average reduction of 13%. Moreover it does not require any changes to the application source code or the OS kernel.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Scheduling

Keywords

Multicore; scheduling; thread migration; lock contention; last-level cache misses

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

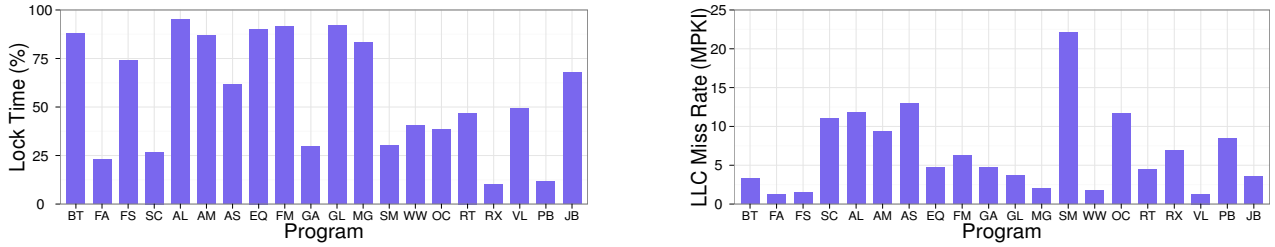
http://dx.doi.org/10.1145/2628071.2628074.

1 Introduction

The cache-coherent multicore multiprocessing architecture was designed to overcome the scalability limits of the symmetric multiprocessing architecture. Today, multicore multiprocessor (or multi-socket) systems with a large number of cores are ubiquitous [4, 5, 6]. For applications with high degree of parallelism it is often necessary to create large number of threads and distribute them across the multiple multicore Sockets to utilize all the available cores [32]. However, shared-memory multithreaded applications often exhibit high *lock times* (i.e., $> 5\%$) due to frequent synchronization of threads [20, 28]. The lock time is defined as the percentage of elapsed time a process has spent waiting for lock operations in user space [26]. Figure 1(a) presents the lock times of 20 multithreaded programs including SPEC jbb2005 [41], PBZIP2 [30], and programs from PARSEC [2], SPEC OMP2001 [41], and SPLASH2 [49] suites. As we can see, the lock times for these programs are quite high.

On a multicore multisocket system, the performance of a multithreaded application with *high lock contention* is highly sensitive to the distribution of threads across Sockets (or multicore processors). In this paper, we demonstrate that the time spent on acquiring locks, as experienced by competing threads, can increase greatly depending upon the Sockets on which they are scheduled. When a lock is acquired by a thread, the time spent on acquiring it is longer if lock currently resides in a cache line of a remote Socket as opposed to the Socket on which the acquiring thread is running. In addition, once the thread acquires the lock, access to shared data protected by the lock is also likely to trigger LLC misses. This causes the thread to spend longer time in the critical section. When the competing threads of a multithreaded application are distributed across multiple Sockets, the above situation arises often and it increases lock transfers between Sockets. Frequent lock transfers between Sockets significantly increases long latency LLC misses that fall on the *critical path* of execution. Moreover, the larger the number of threads, the higher is the lock contention problem which causes LLC misses associated with inter-Socket transfer of locks and the shared data they protect. As shown in Figure 1(b), most of the programs exhibit high LLC miss rates (i.e., > 3 MPKI). LLC miss rate is defined as the last level cache misses per thousand instructions (MPKI). Since the latency of LLC miss is high, the increase in LLC misses on the critical path increases both lock acquisition latency and critical section processing time, thereby leading to significant performance degradation.

PARSEC: bodytrack (BT), fluidanimate (FA), facesim (FS), streamcluster (SC); SPEC OMP2001: applu (AL), ammp (AM), apsi (AS), equake (EQ), fma3d (FM), gafort (GA), galgel (GL), mgrid (MG), swim (SM), wupwise (WW); SPLASH2: ocean (OC), raytrace (RT), radix (RX), volrend (VL); PBZIP2 (PB); SPEC jbb2005 (JB).



(a) Lock times. (b) LLC Miss Rate.

Figure 1: We studied 33 programs including SPEC jbb2005, PBZIP2, and program from PARSEC, SPEC OMP2001, and SPLASH2 suites. We thoroughly evaluate programs with high lock times (i.e., > 5%). Among the 33 programs, the above 20 programs have high lock times. Therefore, we considered these 20 programs throughout this paper. Lock times and LLC miss rates of these 20 programs are shown here. The programs are run with 64 threads on 64 cores spread across 4 Sockets. The description of the machine is shown in Figure 2.

Our Solution -- Shuffling. To address the above problem, we propose a scheduling framework called *Shuffling*, which aims to reduce the variance in the lock arrival times of the threads scheduled on the same Socket. The lock arrival times are the times at which threads arrive at the critical section just before they successfully acquire the lock. *By scheduling threads whose arrival times are clustered in a small time interval so that they can all get the lock without losing the lock to a thread on another Socket.* Thus Shuffling ensures that once a thread releases the lock, it is highly likely that another thread on the same Socket will successfully acquire the lock, and LLC misses will be avoided. Consequently, Shuffling reduces the lock acquisition time and speeds up the execution of shared data accesses in the critical section, ultimately reducing the execution time of the application.

We implemented Shuffling on a 64-core, 4-Socket machine running Oracle Solaris 11 and evaluated it using a large set of multithreaded programs. The results show that for 20 programs with high lock times Shuffling achieves up to 54% (average 13%) reduction in execution time. For the remaining 13 programs with very low lock times, the change in performance was insignificant (within 0.5%). Moreover, Shuffling outperforms the state-of-the-art cache contention management technique [3, 52]. Finally, Shuffling is an attractive approach because its *overhead is negligible* and it *does not require any changes to the application source code or the OS kernel.*

The key contributions of our work are as follows:

- We identify the important reasons for why modern OSs fail to achieve high performance for multithreaded applications with high lock contention running on multicore multiprocessor systems.
- We develop a scheduling framework called *Shuffling* which orchestrates migration of threads between Sockets with the goal of simultaneously maintaining load balance and reducing lock transfers between Sockets to reduce LLC misses on the critical path.

The rest of the paper is organized as follows. In Section 2 we show why distribution of threads impacts lock transfers between Sockets. In Section 3 we provide design and implementation of Shuffling. In Section 4, we evaluate Shuffling with a wide variety of multithreaded programs. Related work and conclusions are given in Sections 5 and 6.

Supermicro 64-core server:
4 × 16-Core 64-bit AMD Opteron™ 6272 (2.1 GHz);
L1/L2: 48 KB / 1000 KB; Private to a core;
L3: 16 MB shared by 16 cores; RAM: 64 GB;
Memory latency (local : remote) = (128 nsec : 201 nsec);
Operating System: Oracle Solaris 11™.

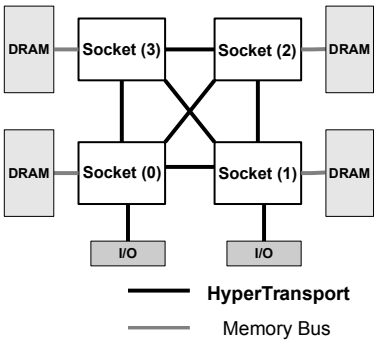


Figure 2: Our 64-core machine has four 16-core Sockets (or Processors). We interchangeably use Socket and Processor.

2 Lock Contention and LLC Misses

Although thread schedulers of modern operating systems, such as Solaris and Linux, are effective in scheduling multiple single threaded programs, they fail to effectively schedule threads of a multithreaded program on a cache-coherent multiprocessor system. Since they do not distinguish between threads representing single threaded applications from threads of a single multithreaded application, the decisions made by these schedulers are oblivious to the lock contention among the threads of a multithreaded program. Lock contention results when multiple threads compete to acquire the same lock. For example, consider a typical synchronization where several threads compete for a lock to enter a critical section in the Bodytrack (BT) [2] computer vision application on the 64 core 4-Socket machine described in Figure 2.

BT is an Intel RMS workload which tracks a 3D pose of a marker-less human body with multiple cameras through an image sequence. BT has a persistent thread pool and the main thread executes the program and sends a task to the thread pool whenever it reaches a parallel kernel. The program has three parallel kernels :1) Edge Detection;

```
// entry of worker thread function
...
// get a lock
ticket = loopTickets.getTicket();
while(ticket < GradientArgs.src->Height() - 2) {
 for(i = ticket; i < GradientArgs.src->Height() -
     2 && i < ticket + WORKUNIT_SIZE_GRADIENT;
     i++) {
       GradientMagThresholdPthread(i + 1, ...);
  }
  ticket = loopTickets.getTicket();
}
...
```

Figure 3: Code snippet around a parallel kernel of BT.

2) Calculate Particle Weights; and 3) Particle Resampling. The parallel kernels use locks to distribute the work among threads to dynamically balance the load. To understand the effect of the distribution of threads across Sockets on the performance, on our multicore system, we studied code around the first parallel kernel Edge Detection. BT employs a gradient based edge detection mask to find edges. The result is compared against a threshold to eliminate spurious edges. Edge detection is implemented in function GradientMagThreshold and the code snippet around the function is shown in Figure 3 [2].

Each worker thread of BT tries to *acquire a lock* so it can enter the critical section. In the critical section the thread runs GradientMagThreshold function in a loop. As expected, lock contention increases with the number of threads. The worker threads perform both read and write operations on the *shared data*. This code involves a series of *successful (Acquire Lock; Execute Critical Section; Release Lock)* operation sequences that are performed by the participating threads. The execution time of an operation sequence increases if LLC misses occur when acquiring the lock and performing read and write operations on the shared data.

2.1 The Cost of LLC Misses

If $T(Acq_i; Exec_i; Rel_i)$ denotes the time for the i^{th} successful operation sequence, and there are a total of n threads competing for the lock, then the total time is given by $\sum_{i=1}^{n} T(Acq_i; Exec_i; Rel_i)$. On a cache-coherent multiprocessor machine, the time for each successful (Acquire; Exec; Release) can vary significantly. Let us consider two consecutive operations $(Acq_i; Exec_i; Rel_i)$ and $(Acq_{i+1}; Exec_{i+1}; Rel_{i+1})$ such that they are performed by two threads located at Sockets $Socket_i$ and $Socket_{i+1}$ respectively. The time for the second operation pair, i.e. $T(Acq_{i+1}; Exec_{i+1}; Rel_{i+1})$ can vary as follows:

- $T(Acq_{i+1}; Exec_{i+1}; Rel_{i+1})$ is *low* (T_{low}) when the threads that perform operations $(Acq_i; Exec_i; Rel_i)$ & $(Acq_{i+1}; Exec_{i+1}; Rel_{i+1})$ are located on the same Socket, i.e. $Socket_i = Socket_{i+1}$ because then LLC hits occur when the lock, and the shared data it protects, are accessed;

- $T(Acq_{i+1}; Exec_{i+1}; Rel_{i+1})$ is *high* (T_{high}) when the threads that perform operations $(Acq_i; Exec_i; Rel_i)$ & $(Acq_{i+1}; Exec_{i+1}; Rel_{i+1})$ are located on different Sockets, i.e. $Socket_i \neq Socket_{i+1}$ because LLC misses occur when the lock, and the shared data it protects, are accessed.

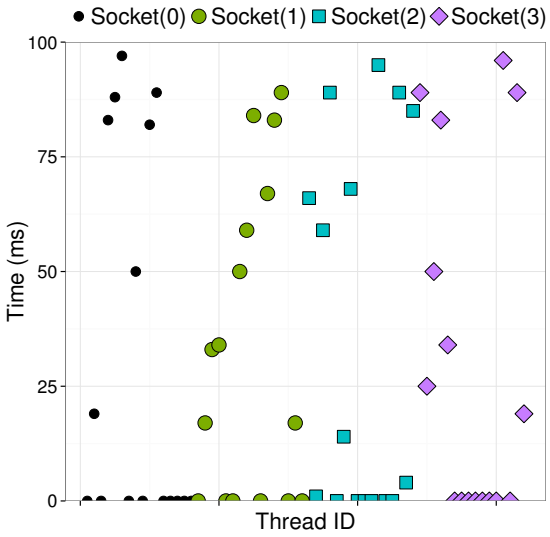


Figure 4: Lock arrival Times of the 64 threads of BT with **Solaris** across all the four Sockets. DTrace [8] scripts were used to collect the lock arrival times of 64 threads of BT before acquiring the lock in a 100 milliseconds interval. Threads with the same symbol arrived on the same Socket.

The above variation is due to the high cost of lock transfer between Sockets on a cache coherent multisocket system. Our machine uses the MOESI cache coherency protocol [42]. In this protocol a cache line can be in one of five states: *Modified, Owned, Exclusive, Shared, and Invalid*. Therefore, following the execution of $(Acq_i; Exec_i; Rel_i)$, the cache line containing the lock is in *Modified* state. If the $Socket_{i+1}$ is different from $Socket_i$, then the cache line must be transferred from $Socket_i$ to $Socket_{i+1}$ when $(Acq_{i+1}; Exec_{i+1}; Rel_{i+1})$ is executed.

2.2 High Frequency of LLC Misses and its Cause

Next, we show how often the above event sequences entail overhead of T_{low} vs. T_{high}. For a given number of threads, the total number of *successful* $(Acquire; Exec; Release)$ operations performed is the same. However, how often these operations require T_{low} and T_{high} overhead varies based on the distribution of threads across the four Sockets. When running BT with 64 threads on 64 cores across 4 Sockets with the default Solaris scheduler, around 68% of the time lock transfers between two Sockets take place. In other words, T_{high} cost must be paid for over $2/3^{rd}$ of the time with the default Solaris scheduler. Therefore, with Solaris, high overhead LLC misses happen frequently which degrades performance as these LLC misses are on the critical path. LLC miss has high latency, for example, it is 201 nanoseconds for our machine. These high latency LLC misses on the critical path significantly increase both *lock acquisition latencies* and *critical section processing times*.

The frequent lock transfers between Sockets in multiprocessor system increases execution time of the critical path and significantly degrades performance of multithreaded applications. This problem becomes very prominent for applications with high lock contention. As Figure 1 shows, BT experiences high lock contention. Its LLC miss rate is also high – 3.3 MPKI (Misses Per Thousand Instructions).

Next let us examine the reason for frequently encountered LLC misses. Figure 4 shows the times (say *lock arrival times*) at which threads arrive at the critical section just before they successfully acquire the lock and execute the event sequence (*Acquire*; *Execute*; *Release*). The x-axis plots thread ids and the y-axis plots the corresponding lock arrival times. We use DTrace [8] scripts to collect the *lock arrival times* of the 64 threads in a 100 milliseconds (ms) snapshot. As we can see in Figure 4, the lock arrival times of threads on each Socket are spread across a wide time interval for the default Solaris scheduler. For example, as we can see in Figure 4, on Socket(0), a thread arrived at the critical section at around 20 ms timestamp, but no other thread arrives at the critical section on Socket(0) till the 50 ms timestamp. Therefore, even though the lock is available before the 50 ms timestamp, none of the threads have arrived at the critical section on Socket(0) to acquire it until the 50 ms timestamp. Therefore, the likelihood that the lock will be acquired before the 50 ms timestamp by a thread on a different Socket is very high. This will trigger lock transfer, along with the shared data protected by the lock, between Sockets. Thus LLC misses will occur due to lock acquisition as well accesses to shared data in the critical section. Since these LLC misses are on the critical path, significant degradation in performance results.

3 The Shuffling Framework

To address the above problem, we propose a scheduling framework called *Shuffling*, which aims to reduce the variance in the arrival times of the threads scheduled on the same Socket. *By scheduling threads whose arrival times are clustered in a small time interval so that they can all get the lock without losing the lock to a thread on another Socket.* Therefore, Shuffling ensures that once a thread releases the lock it is highly likely that another thread on the same Socket will successfully acquire the lock and LLC misses will be avoided. Consequently, Shuffling reduces the lock acquisition time and speeds up the execution of shared data accesses in the critical section, ultimately reducing the execution time of the application.

Let us assume that the expected arrival times at critical section for all the threads are known. We can distribute the threads across the Sockets as follows. We sort the threads according to their expected arrival times and divide them into equal sized groups, one group per Socket to maintain load balance across the Sockets. Moreover, each group can be formed by taking consecutive threads from the sorted list to minimize the variance in arrival times for threads in each group. Finally, by migrating the threads, we distribute the threads across the Sockets according to the above schedule.

We observe that, instead of moving locks and shared data they protect between the Sockets, Shuffling moves the threads between the Sockets. *Migrating threads between Sockets is preferable to moving locks, and shared data they protect, between Sockets.* This is because when multiple threads are contending for locks and shared data, they are not doing useful work and hence the thread migration cost is not expected to impact the execution time. On the other hand, Shuffling reduces LLC misses on the critical path of execution. Moreover, it preserves the number of threads on each Socket, i.e. *load balance* across the Sockets is maintained. We also observe, that it reduces inter-Socket thread migrations compared to default Solaris. Thus shuffling improves application performance.

Algorithm 1: The Shuffling Framework.

Input: N: Number of threads; C: Number of Sockets.

repeat
  i. Monitor Threads -- sample lock times of N threads.
  if *lock times exceed threshold* **then**
    ii. Form Thread Groups -- sort threads according to lock times and divide them into C groups.
    iii. Perform Shuffling -- shuffle threads to establish newly computed thread groups.
  end
until *application terminates*;

The key problem that we face is -- how to determine the expected arrival times of threads. One way to achieve this is to analyze the application code and instrument it to collect this information. This task is complicated by the fact that an application typically contains many critical sections and it would require analyzing the application before it can be run to take advantage of Shuffling. However, *we would like to develop a Shuffling algorithm that does not modify application code and can be applied on-the-fly to any application that is run on the system.* Therefore we base our shuffling algorithm on thread *lock times* that can be collected by using simple OS level monitoring utilities. As we show later in this section, sorting and grouping of threads according to their *lock times* is effective in reducing the variance in the arrival times of the threads scheduled on each Socket.

3.1 The Shuffling Algorithm

The overview of Shuffling is provided in Algorithm 1. Shuffling is implemented by a daemon thread which executes throughout an application's lifetime repeatedly performing the following three steps: monitor threads; form thread groups; and perform thread shuffling. The first step monitors the behavior of threads in terms of the percentage of elapsed time they spend on waiting for locks. If this time exceeds a preset threshold (we use 5% of elapsed time as the threshold), shuffling is triggered by executing the next two steps. The second step forms groups of similarly behaving threads using the lock times collected during the monitoring step. Finally the third step performs thread shuffling to ensure that threads belonging to the same thread group are all moved to the same CPU. Next we describe these steps in greater detail.

(i) Monitor Threads. We monitor the fraction of execution time that each thread spends waiting for locks in user space -- this is referred to as the *lock time*. *Threads that experience similar lock times will be placed in the same group* as they are likely to represent threads that contend with each other for locks and keep the lock located on the same Socket. The daemon thread maintains per thread data structure that holds the lock time values collected for the thread as well as the id of the Socket on which the thread is running. The design of the monitoring component involves two main decisions: a) selecting an appropriate sampling interval for lock times and b) selecting an appropriate shuffling interval i.e., time interval after which we form new thread groups and carry out shuffling. We explain the selection of these intervals in Section 3.2.

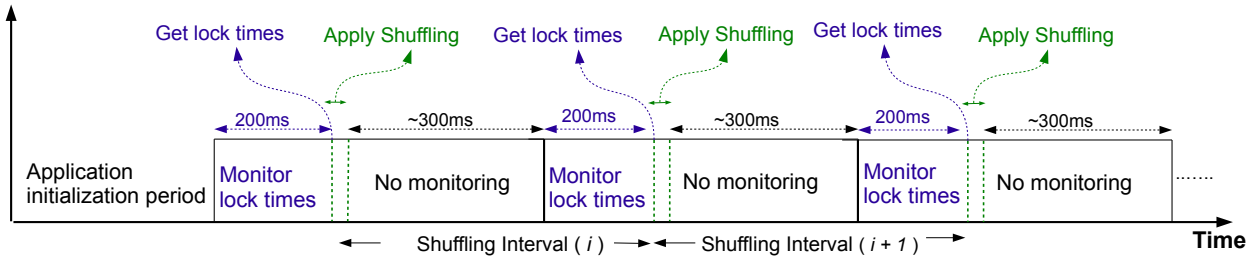


Figure 5: Frequency of Monitoring and Shuffling.

(ii) Form Thread Groups. At regular intervals (i.e., *shuffling interval*), the daemon thread examines the profile (i.e., lock time) data collected for the threads, and if the lock times exceed the minimum preset threshold, it constructs thread groups such that one group per Socket is formed. We would like the lock time behavior of threads within each group to be similar. Therefore we sort the threads according to their lock times and then divide them into as many groups of consecutive threads as the Sockets being used to run the application. For example, when all Sockets are being used, the first group is assigned to Socket(0), the second to Socket(1), the third to Socket(2), and the fourth to Socket(3). Since the size of each thread group is the same, load balance across the Sockets is maintained.

(iii) Perform Shuffling. This step simply affects the thread groups computed for each of the Sockets in the preceding step. That is, at regular intervals (i.e., shuffling interval), Shuffling simultaneously migrates as many threads as needed across Sockets to realize the new thread groups computed in the previous step [1]. In this step we expect only a subset of threads to be migrated as many threads may already be bound to a set of cores on the Sockets where we would like them to be. It is also possible that in some programs, over many shuffling intervals, the behavior of threads does not change significantly. If this is the case, the thread groups formed will not change, and hence no threads will be migrated. Thus, effectively the shuffling step is skipped altogether and monitoring is resumed. In other words, when thread migrations are not expected to yield any benefit, they will not be performed.

3.2 Frequency of Monitoring and Shuffling

In developing a practical implementation of the Shuffling framework we must make several *policy* decisions. First, we must select an appropriate *sampling interval* for collecting lock time data of the threads before thread groups can be formed for shuffling. Second, we must select an appropriate *shuffling interval*, i.e. duration between one application of shuffling and the next.

3.2.1 Sampling Interval.

We monitor the lock times of threads through the proc file system. Although small sampling interval allows fine-grain details of the lock time data to be collected, it increases the monitoring overhead. Therefore selecting an appropriate sampling interval is very important. To select an appropriate sampling interval, we measured the system overhead of Shuffling with different lock time sampling intervals when programs are running with 64 threads on our machine. When we use time intervals less than 200 milliseconds (ms) (e.g., 50 ms, 100 ms), the system overhead is significantly higher. This is due to the high rate of cross-calls, which lead to high system time [26]. Therefore, the Shuffling framework uses 200 ms time interval for collecting lock times. The overhead of monitoring lock times with 200 ms is *less* than 1% of system time, which is negligible.

[1]For balancing load across cores, Shuffling delegates the job of migrating threads *within cores* of a Socket to the default Solaris thread scheduler.

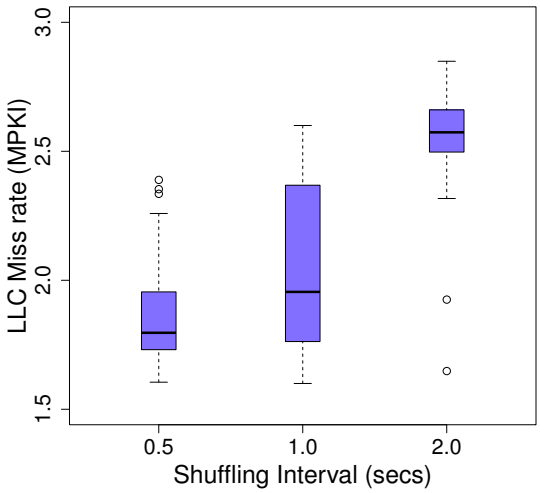


Figure 6: LLC Miss Rates vs Shuffling Interval: shuffling with 500 milliseconds interval provides sustained reduction in LLC miss rate.

3.2.2 Shuffling Interval.

Shuffling interval significantly impacts the performance improvements as it affects lock transfer rate between Sockets, and thus affects LLC misses in the critical path. Therefore, for finding an appropriate shuffling interval, we measured the LLC miss-rate with different shuffling intervals as shown in Figure 6 by running BT with 64 threads on 64 cores with the Shuffling framework. As we can see in Figure 6, shuffling with 500 ms interval provides sustained reduction in LLC miss rate and gives best performance for BT. Although we can shuffle with an interval less than 500 ms by using small lock time sampling intervals, as explained above, it increases system overhead and limits performance benefits. Therefore, we have chosen 500 ms as a shuffling interval in this work.

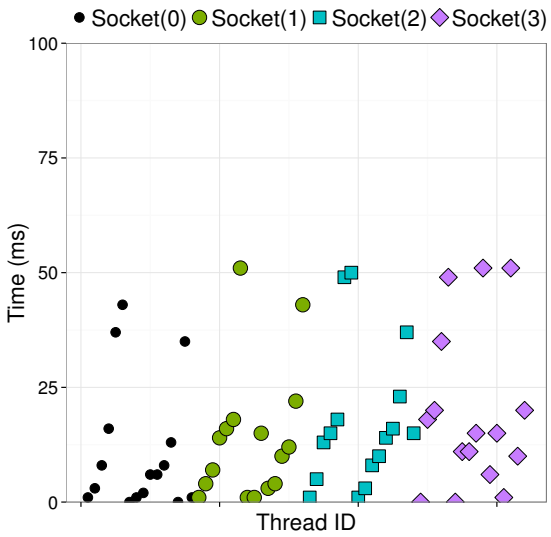


Figure 7: Lock arrival Times of the 64 threads of BT with **Shuffling** across all the four Sockets. DTrace scripts were used to collect the arrival times of 64 threads of BT before acquiring the lock in a 100 milliseconds interval.

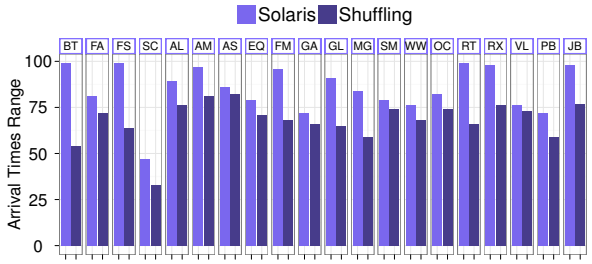


Figure 8: Lock arrival times ranges with Solaris vs. Shuffling.

As shown in Figure 5, for every 500 ms, the Shuffling framework collects lock times using prstat(1) utility with 200 ms sampling interval, constructs thread profile data structures with the lock times and core ids, form groups, and then assigns each of the groups to a set of cores (or Sockets). It repeats this process until completion of the program. Finally, the pset_lwp_bind(2) system call is used for binding a group of threads to a set of cores (called a processor-set in Solaris terminology). A processor-set is a pool of cores such that if we assign a multithreaded application to a processor-set, then during load balancing, the OS restricts the migration of threads across the cores within the processor-set [26].

3.3 Impact of Shuffling on Bodytrack (BT)

We show that shuffling based on thread lock time measurements does have the intended effect, i.e. reduction in variance across arrival times of threads on a given Socket. Unlike the lock arrival times of BT with Solaris (Figure 4 in Section 2), as we can see in Figure 7, the lock arrival times of threads are closely clustered minimizing variance when Shuffling is employed. Moreover, Figure 8 shows that the range over which lock arrival times of threads are distributed reduces significantly when Shuffling is used.

Table 1: BT achieves 54% performance improvement.

	Solaris	**Shuffling**
T_{max} Lock Transfers	68%	43%
LLC Miss Rate	3.3 MPKI	1.9 MPKI
Lock Time	88%	74%
Avg. Execution Time	113 secs	52 secs

When running BT with 64 threads on 64 cores across 4 Sockets with Shuffling, only around 43% of the time the lock transfers take place between the local Socket and a remote Socket. In other words, T_{max} lock transfers are encountered roughly 2/5th of the time. However, as we described in Section 2, with Solaris, T_{max} lock transfers are encountered roughly 2/3rd of the time. Therefore, Shuffling dramatically reduces high overhead lock transfers and thus, reduces LLC misses on the critical path. While LLC miss rate of BT with Solaris is 3.3 MPKI, LLC miss rate with Shuffling is 1.9 MPKI. Moreover, lock time of BT with Solaris is 88%, lock time with Shuffling is 74%. That is, with Shuffling, there is 42% reduction in LLC miss rate and lock time reduces by 14%. BT achieves a large performance improvement (54%) with Shuffling. Table 1 summarizes these observations.

The large improvement in performance of BT is in part due to additional contributing factors: reduction in coherence traffic and inter-Socket thread migrations. By minimizing coherence traffic, Shuffling reduces latency of LLC miss. Moreover, by keeping threads close to locks, Shuffling reduces lock time; thus, threads go to sleep state far less often with Shuffling compared to Solaris. We observed that when a thread made a transition from sleep state to a ready state, often the core on which it last ran is not available and it is migrated to a core available on another Socket. Therefore, Shuffling encounters far fewer thread migrations between Sockets than Solaris -- we observed that while the inter-Socket thread migration rate with Solaris was 270 migrations per second, it is only 16 migrations per second with Shuffling.

We have explained how Shuffling improves the performance of the BT multithreaded program. In the next section, we present its evaluation with a wide variety of multithreaded programs.

4 Evaluation

We demonstrate the merits of Shuffling via experimental evaluation of the performance of several multithreaded applications. We also provide a detailed explanation of the overheads associated with Shuffling. We seek answers of the following questions using these experiments:

- Does Shuffling achieve better performance for multithreaded programs than modern OS such as Solaris which is oblivious of the lock contention among multiple threads belonging to an application?
- How does Shuffling perform compared to the state-of-the art cache contention management technque [3] and PBind (pinning one thread to core)?
- Is Shuffling effective in running multiple multithreaded programs simultaneously on multisocket systems?

We compare Shuffling with the following:

(i) Solaris -- the default Oracle Solaris 11.

(ii) DINO[3] is a state-of-the-art cache contention management technique. It reduces cache contention (LLC miss rate) by separating memory intensive threads by scheduling them on different sockets of a multicore machine. By combining low memory-intensive threads with high memory-intensive threads, the cache pressure is balanced across sockets and thus overall cache miss-rate is reduced.

(iii) PSets only permits intra-socket migration of threads, i.e. threads can migrate from one core to another on the same Socket. PSets scheduling essentially divides threads into four groups at the beginning of execution and assigns them to four Sockets. That is assigning threads 1 to 16 to Socket(0), threads 17 to 32 to Socket(1), threads 33 to 48 to Socket(2), and threads 49 to 64 to Socket(3). PSets neither changes the groups of threads nor migrate threads between the groups.

(iv) PBind is nothing but the commonly used one-thread-per-core Binding (or pinning) model, i.e. it does not allow any thread migrations. Using pbind(1) utility, we bind 64 threads to 64 cores, one thread per core. The difference between PSets and PBind is that thread migrations are allowed *within the groups (or processors)* with Psets.

4.1 Experimental Setup

Our experimental setup consists of a 64-core machine running Oracle Solaris 11. Figure 2 shows its configuration. We have chosen Solaris OS as its Memory Placement Optimization feature and Chip Multithreading optimization allow to effectively support multicore multiprocessor systems with large number of cores. Specifically, Solaris kernel is aware of the latency topology of the hardware via locality groups that guides scheduling and resource allocation decisions. It provides several effective low-overhead observability tools (e.g., DTrace [8]). Solaris uses next touch memory allocation policy [26]. All programs considered use pthreads and pthreads use adaptive_mutex locks (spin-then-block policy) [21].

Benchmarks. We evaluate Shuffling using a wide variety of 33 multithreaded programs including SPEC jbb2005, PBZIP2, and programs from PARSEC, SPEC OMP2001, and SPLASH2 suites. The implementations of PARSEC programs and SPALSH2 are based upon *pthreads* and we ran them using native inputs (i.e., the largest inputs available). SPEC OMP2001 programs were run on medium sized inputs. SPEC jbb2005 (JBB) with single JVM is used in all our experiments.

We present detailed performance data for 20 programs of Figure 1. These programs have substantial parallelism and high lock times. For the remaining 13 programs (of all the 33 programs) with low lock times (i.e., less than 5%), data is presented to simply show that their performance is unaffected by Shuffling. We also exclude programs with very short running times (< 10 seconds).

Performance Metrics. We ran each experiment 10 times and present average and coefficient of variation (CV) results from the 10 runs. Coefficient of variation is defined as the ratio of the standard deviation to the mean. The performance metrics we use are: *percentage reduction in running time* for all the programs except SPECjbb2005 (*improvement in throughput*). We ran programs with 64 threads to use all four Sockets in all these experiments.

Table 2: Performance improvements relative to Solaris. Shuffling improves performance up to 54% and an average of 13%. Shuffling significantly outperforms DINO, PSets, and Pbind.

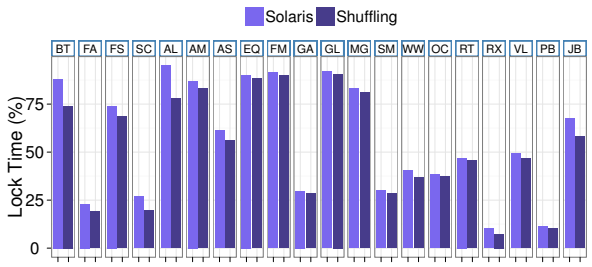
	Performance Improvement (%)			
Program	**Shuffling**	**DINO**	**PSets**	**PBind**
BT	54.1	-5.0	-2.0	-3.3
FA	6.0	2.1	7.0	-12.7
FS	12.0	-13.1	1.1	1.0
SC	29.0	6.0	-13.0	-15.0
AM	9.3	-11.0	1.0	-115.0
AS	13.0	-21.0	4.6	-2.3
AL	13.2	-17.0	-1.0	-95.0
EQ	9.0	-6.0	-0.1	-2.3
FM	10.7	-5.8	2.0	1.0
GA	4.0	-9.0	-0.1	2.5
GL	9.1	-5.1	0.1	-4.6
MG	8.8	-8.0	1.0	-11.8
SM	4.7	-5.0	0.1	1.9
WW	5.2	-6.0	-2.0	-17.2
OC	13.4	3.1	4.2	2.1
RT	4.0	-10.8	2.0	-4.0
RX	19.0	5.0	-13.0	-7.4
VL	12.8	-7.0	2.8	-1.0
PB	13.0	11.2	-12.9	-21.0
JB	14.0	-2.0	-1.0	-7.4

4.2 Performance Benefits

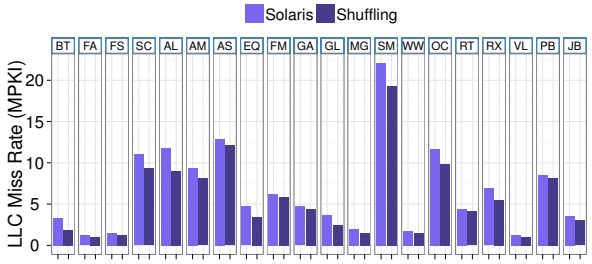
As we can see in Table 2, Shuffling significantly outperforms default Solaris as well as all other techniques. Substantial performance improvements (13% on average) are observed over the default Solaris. For the *bodytrack (*BT*)* program the improvement is the highest -- around 54%. The throughput of SPEC jbb2005 (JB) is improved by 14%. The programs that exhibit high lock contention get high performance improvements with Shuffling. PSets is slightly better than DINO and it also outperforms Shuffling for the low memory intensive and low lock contention program, FA. Shuffling achieves low performance improvements for the programs (e.g., SM) with large working sets. This is because of the tradeoff between sacrificed cache locality and improved lock acquisition times.

However, both PSets and DINO give worse performance for several programs even when compared to the default Solaris scheduler. DINO outperforms Solaris for programs with low lock contention -- programs PB, RX, FA, and SC. Compared to Solaris, DINO gives better performance for 25% of the programs and PSets for 40% of the programs. However, the default Solaris scheduler is better on average compared to both DINO and PSets scheduling techniques. Though DINO is effective for a mix of *single threaded* workloads where half of the threads are memory-intensive and other half are CPU-intensive, it may not work well for multithreaded programs with *high lock contention*. One can view Shuffling and DINO as complementary techniques that can be potentially combined to further improve performance -- using DINO for lock contention free, but memory intensive programs.

Figure 9 shows lock times and LLC miss rates of the 20 programs. As we can see, Shuffling reduces both the lock times and LLC miss rates compared to Solaris, and thus improves performance. Moreover, as we can see in Figure 10,



(a) Lock times.



(b) LLC miss rates.

Figure 9: Shuffling reduces both lock times and LLC misses.

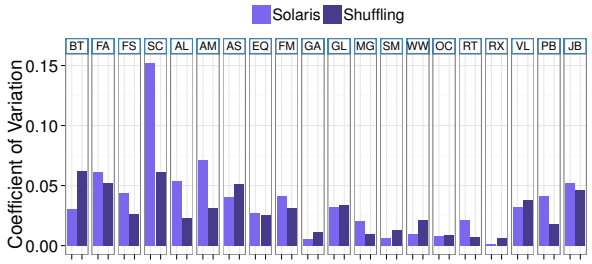


Figure 10: Shuffling reduces performance variation.

the performance variation is low (coefficient of variation $<$ 0.1) for most of the programs with both Solaris and Shuffling.

4.2.1 Why PBind Degrades Performance?

As we can see in Table 2, PBind (pinning one thread per core) significantly degrades performance of several programs compared to the other algorithms. Specifically, PBind gives poor performance for programs with high lock contention on multicore multiprocessor systems with a large number of cores. As the threads of a multithreaded program do not migrate across Sockets of a multicore multiprocessor system with PBind, the frequency of high overhead lock transfers between Sockets increases compared to the default Solaris. This leads to an increase of LLC misses on the critical path of the multithreaded program, which leads to degraded performance [34].

4.2.2 Shuffling the remaining 13 programs.

While the above detailed performance data is for the 20 programs with high lock times, we also applied Shuffling to the remaining 13 programs (of all the 33 programs). We divide these 13 programs into two groups: 1) with low lock times and 2) with very short running times. The programs in the first group have low lock times (less than 5% of the

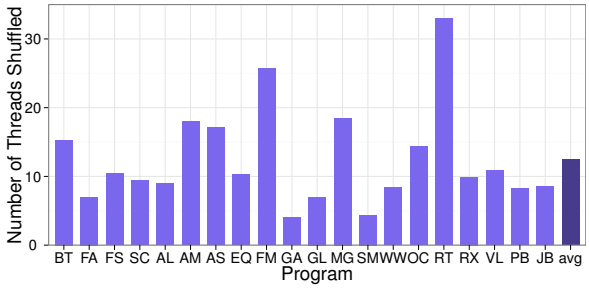


Figure 11: The degree of Shuffling.

execution time) because the serial part of the execution, executed by the main thread, accounts for 90% to 95% of the execution time -- the programs are blackscholes, dedup, canneal, ferret, x264, vips, and raytrace from PARSEC. The purpose of this experiment was to determine if Shuffling can hurt performance when it is applied to programs where it is not needed. We observed that the change in execution time was insignificant, i.e. less than 0.5%. This is not surprising as, when lock times are small, the cost of Shuffling is simply the cost of monitoring as no thread migrations are triggered. The programs in the second group have very short running times (between 2 secs and 5 secs) -- the programs are barnes, fft, fmm, lu, radiocity, and water from SPLASH2.

4.3 Overhead of Shuffling

Now let us consider the overhead of Shuffling. The cost of monitoring thread lock times is very low – around 1% of the CPU utilization can be attributed to the monitoring task. The cost of migrating threads during shuffling is also small. The bar graph in Figure 11 shows the average number of threads shuffled (across Sockets) in a single shuffle operation for each of the programs. This number ranges from a minimum of 4 threads for GA to a maximum of 32 threads for RT. Across all the programs, the average is 12.4 threads being migrated during each shuffling operation. Since the total number of threads is 64, this represents around 19% of the threads. The system call for changing the binding of a single thread from cores in one CPU to cores in another CPU is around 29 microseconds. Therefore every 500 milliseconds Shuffling spends around 360 microseconds (29×12.4) on changing the binding of migrated threads. Thus, this represents 0.0007% of the execution time. Therefore, the overhead of Shuffling is negligible.

4.4 Time Varying Behavior of Degree of Shuffling

Finally we study the time varying behavior of thread shuffling as observed throughout the executions of the programs. In Figure 12 the number of threads that are shuffled is plotted over the entire execution of the programs. We see several different types of behavior if we examine the behavior following the startup period, i.e. the initial period of execution. For some benchmarks (BT, FA, AM, AS, MG, RT, RX, VL, WW) the execution can be divided into *small number of intervals* such that during each interval the number of threads shuffled *varies within a narrow range*. For a few benchmarks (FS, AL, SC, EQ, GA, GL, SM) the number of threads shuffled *varies rapidly within a narrow range* after the initial execution period. For some programs (PB, JB) the number of threads shuffled *varies rapidly across a wide range*. Finally, for some programs *often no threads are migrated* – (GA, GL, SC, SM).

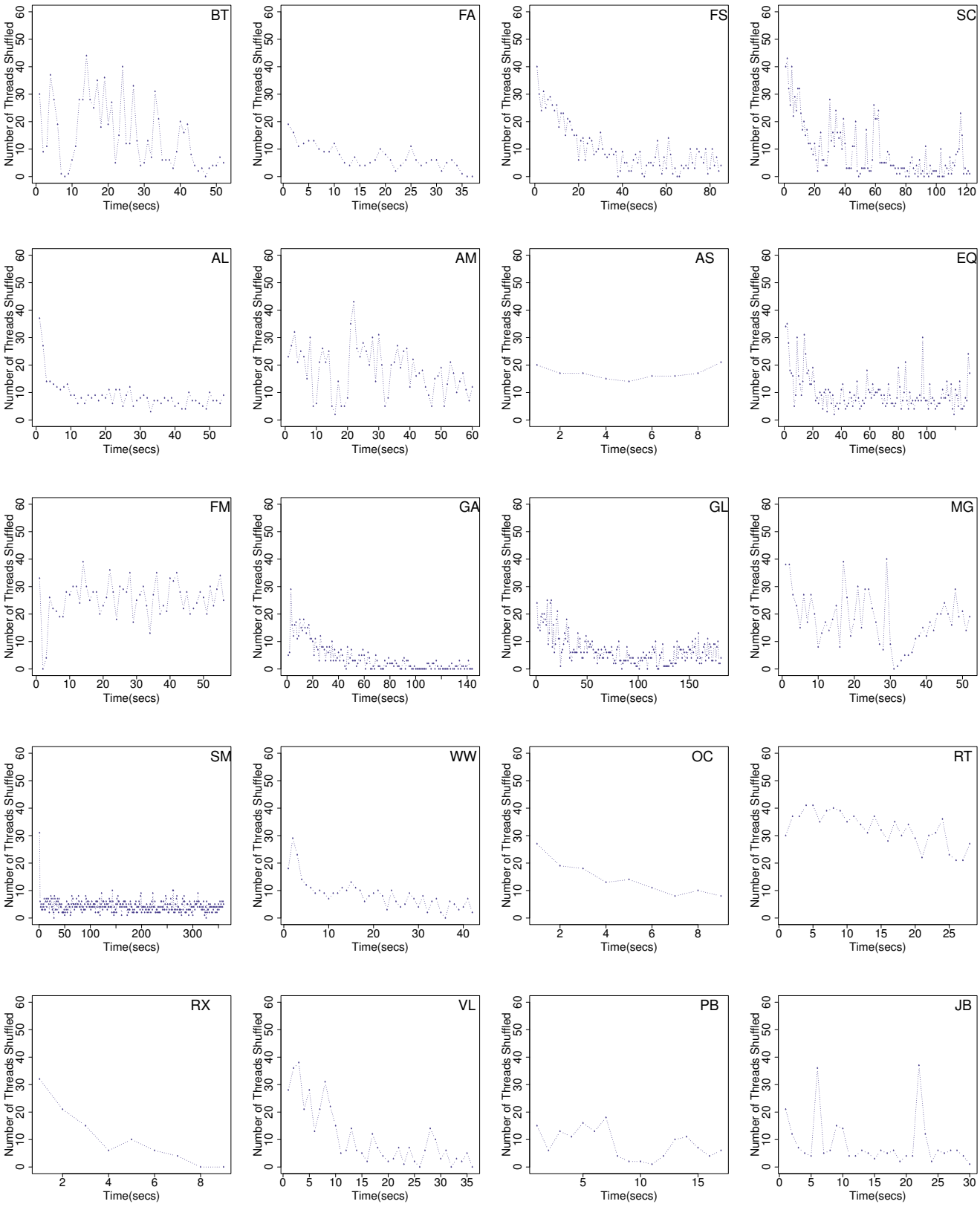


Figure 12: Time varying behaviour of degree of thread shuffling. The number of threads that are shuffled every 500 milliseconds is plotted over the entire execution of the programs.

Table 3: Shuffling multiple applications. Running times of the programs are expressed in seconds (s).

	FS	MG	AL	AM	FS	AL
Solaris	106s	74s	71s	72s	104s	69s
Shuffling	83s	67s	62s	65s	82s	62s
Improvement	21.7%	9.5%	12.7%	9.7%	21.2%	10.1%

4.5 Multiple Applications

So far we have considered performance of a single application being run on the system. In practice multiple multithreaded applications may be run simultaneously on the system. Next we ran pairs of programs, each using all 4 Sockets, to see if Shuffling improves performance of both applications. We ran a pair of compute intensive applications (FS and MG); a pair of memory intensive applications (AL and AM); and a combination of compute (FS) and memory (AL) intensive applications. The results of running these application pairs with Solaris 11 and Shuffling are presented in Table 3 (s next to the times indicates seconds). As we can see, in all three cases, both applications benefit from Shuffling and their performance improvements are: 21.7% and 9.5%; 12.7% and 9.7%; and 21.2% and 10.1%. Therefore we conclude that Shuffling is robust because it is beneficial for the multiple applications that are running on each Socket simultaneously.

5 Related Work

There are many aspects of the lock contention problem and thus range of strategies have been proposed to address lock contention. Three specific approaches proposed in prior work include: *thread migration* [19, 40]; *thread clustering* [50]; and *contention management* [50, 21].

5.1 Thread Migration Techniques

The work most closely related to our work is RCL [19] in which authors make the observation that most multithreaded applications do not scale to the number of cores found in modern multicore architectures, and therefore it may be beneficial to dedicate some of the cores to serving critical sections. While this technique works well for some multithreaded applications, it has the drawback that application must be modified -- critical sections must be identified and reengineered [19]. However, Shuffling does not modify application code and can be applied *on-the-fly* to any application that is run on the system. Moreover, it is also effective for scheduling multiple multithreaded applications and its overhead is negligible.

Another work closely related to our work is [40] in which authors make the observation that, in multicore multiprocessor systems, it may be beneficial to employ thread migration to reduce the execution time cost due to acquiring of locks. They propose a migration technique that is incorporated in the OS thread scheduler. While this technique performs well for a few microbenchmarks [40], for majority of SPLASH2 programs the technique frequently yielded large performance degradation. Our Shuffling technique is superior to [40] as it consistently provided performance improvements across a large set of multithreaded programs including SPLASH2 programs. [13] uses an online model to determine how many cores to allocate to lock intensive programs to reduce lock contention in the kernel space. The idea is to separate lock-intensive kernel tasks from lock-free kernel tasks and control kernel lock contention by allocating appropriate number of cores. Shuffling focuses on reducing lock contention in the user space by adaptively migrating threads across sockets of multicore multi-CPU system. Unlike [13], Shuffling is effective for coscheduling multiple applications.

Cache contention aware scheduling techniques also employ thread migration across Sockets. These techniques are guided by the last-level cache miss-rates [3, 52, 1, 22, 24, 25, 44, 45, 29, 46, 51, 31]. While these techniques are highly effective for workloads consisting of multiple single threaded programs, they are not effective in scheduling threads of a multithreaded program on a cache-coherent multicore multiprocessor system. This is because they do not consider lock contention among the threads of a program while making thread scheduling decisions. Unlike the above, we demonstrate that Shuffling significantly outperforms these techniques and also it very effective in scheduling of threads of multiple multithreaded applications. [16, 35] presents load balancing techniques to improve performance of multithreaded programs.

5.2 Thread Clustering Techniques

A number of thread clustering techniques have been developed to improve program performance [50, 47, 43]. Of these, the technique in [50] is aimed at reducing the overhead caused by lock contention. This is achieved by clustering threads that contend for the same lock and then schedule them on the same processor. The number of threads in a cluster can be large, in fact all threads in an application will be in the same cluster if they are synchronizing on a barrier. Thus, load is no longer balanced, and parallelism is sacrificed. The authors state "it is possible that the performance benefit of reducing contention is greater than the performance benefit of higher parallelism". In contrast, our Shuffling framework maintains load balance and thus does not sacrifice parallelism. Finally, while the approach presented in [50] is effective for server workloads, our approach is more relevant for highly parallel applications.

The clustering techniques in [47, 43] have a different objective. In [47] authors examine thread placement algorithms to group threads that share memory regions together onto the same processor so as to maximize cache sharing and reuse. It is assumed that the shared-region information is known a priori, i.e. this information is not ascertained dynamically. In [43], Tam et al. propose a thread clustering technique to detect shared memory regions dynamically. In contrast, we focus on minimizing lock transfers between Sockets, and thus minimizing *critical* LLC misses of applications running on a multicore multiprocessor machine with a large number of cores.

5.3 Contention Management Techniques

Another cause of performance degradation is *lock-holder* thread preemptions as they slow down the progress of thread holding the lock. In [50] a technique is presented for reducing lock holder preemptions. While it is important to avoid lock-holder preemptions, this technique is complementary to our approach. Preventing lock-holder preemptions reduces the duration for which a thread *holds* the lock while our thread shuffling techniques reduce the time it takes for a thread to *acquire* a lock. [21, 33] propose a load control mechanism to decouple load management from lock contention management. This approach uses blocking to control the number of runnable threads and then spinning in response to

contention. While above techniques are aimed at contention management, our work is aimed at reducing the overhead of contention that manifests after the management techniques have been applied.

5.4 Other Works

Complementary to our work are improved locking and synchronization algorithms. In [14, 15, 36] authors propose new locking mechanisms. In contrast, our work focuses on adapting the location of threads across the Sockets of a cache-coherent multicore multiprocessor machine to reduce the lock transfers between Sockets and thus critical LLC misses. In [10] NUMA's impact on performance of different barrier synchronization algorithms is studied but no solutions are proposed. Several researchers [7, 23, 11, 48, 38, 12, 27, 18] studied the impact of NUMA on the performance of parallel applications and developed optimizations and adaptive scheduling techniques. Gupta et al. [18] explored the impact of the scheduling strategies on the caching behavior. Chandra et al. [9], evaluate different scheduling and page migration policies on a cache-coherent multiprocessor system. Sasaki et al. [37] developed a scheduling technique for allocating optimal number of cores to multithreaded programs. None of the above works consider lock-access latency and critical section processing times of threads for effective scheduling them on multicore machines.

6 Conclusions

We demonstrated that the performance of a multithreaded application with high lock contention running on a multicore multiprocessor system is very sensitive to the distribution of threads across multiple Sockets. To address this problem, we presented Shuffling that reduces transfer of locks between Sockets by migrating threads of a multithreaded program across Sockets such that threads seeking locks are more likely to find the locks on the same Socket. Migrating threads between Sockets is preferable to moving locks between Sockets. This is because when multiple threads are contending for locks and shared data, they are not doing useful work and hence the thread migration cost is not expected to impact the execution time. However, lock transfer times are on the critical path of execution and hence preventing lock transfers between Sockets reduces execution time. We evaluated Shuffling with a wide variety of multithreaded programs and the experimental results show that Shuffling achieves up to 54% reduction in execution time and an average reduction of 13%. Moreover it has very low overhead and it does not require any changes to the application source code or the OS kernel.

7 Acknowledgments

This work is supported by National Science Foundation grants CCF-1157377, CCF-0963996, CCF-0905509, and CSR-0912850 to the University of California Riverside.

8 References

[1] M. Bhadauria and S. A. McKee. An Approach to Resource-Aware Co-Scheduling for CMPs. In *ICS*, 2010.

[2] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[3] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-aware Contention Management on Multicore Systems. In *USENIX ATC*, 2011.

[4] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek. Reinventing Scheduling for Multicore Systems. In *HotOS*, 2009.

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

[6] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, F. Kaashoek, R.Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *OSDI*, 2010.

[7] T. Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *SEDMS*, 1993.

[8] B. Cantrill, M. Shapiro, and A. Leventhal. Dynamic instrumentation of production systems. In *USENIX ATC*, 2004.

[9] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *ASPLOS 1994*.

[10] J. Chen, W. Watson, and W. Mao. Multi-Threading Performance on Commodity Multi-core Processors. In *HPC-Asia*, 2007.

[11] J. Corbalan, X. Martorell, and J. Labarta. Evaluation of the Memory Page Migration Influence in the System Performance: the Case of the SGI O2000. In *SC*, 2003.

[12] J. Corbalan, X. Martorell, and J. Labarta. Performance-driven processor allocation. In *OSDI*, 2000.

[13] Y. Cui, Y. Wang, Y. Chen, and Y.Shi. Lock-contention-aware scheduler: A scalable and energy-efficient method for addressing scalability collapse on multicore systems. In *ACM TACO*, 4, Article 44, Jan. 2013.

[14] D. Dice, V. Marathe, and N. Shavit. Flat Combining NUMA Locks. In *SPAA*, 2011.

[15] D. Dice, V. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *PPoPP*, 2012.

[16] X. Ding, K. Wang, P.B. Gibbons, and X. Zhang. BWS: balanced work stealing for time-sharing multicores. In *Eurosys*, 2012.

[17] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez. Adaptive parallel job scheduling with flexible coscheduling. In *IEEE TPDS*, (2005), 16(11).

[18] A. Gupta, A. Tucker, and S. Urushibara. The Impact Of Operating System Scheduling Policies And Synchronization Methods On Performance Of Parallel Applications. In *SIGMETRICS*, 1991.

[19] L. Jean-Pierre, D. Florian, T. Gaël, L. Julia and M. Gilles. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX ATC*, 2012.

[20] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck Identification and Scheduling in Multithreaded Applications. In *ASPLOS*, 2012.

[21] R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. In *ASPLOS*, 2010.

[22] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. In *IEEE Micro*, 2008

[23] R.P. Larowe, C. S. Ellis, and M. A. Holliday. Evaluation of NUMA Memory Management Through Modeling and Measurements. In *IEEE TPDS*, (1991), 688 -- 701.

[24] Z. Majo and T. R. Gross. Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead. In *ISMM*, 2011.

[25] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, 2011.

[26] R. McDougall and J. Mauro. Solaris Internals. Prentice Hall Publications, Second Edition, 2006.

[27] R. McGregor, C. Antonopoulos, and D. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *IPDPS*, 2005.

[28] A. Mendelson and F. Gabbay. 2001. The effect of seance communication on multiprocessing systems. In *ACM Trans. Comput. Syst.* 19, 2 (May 2001), 252-281.

[29] A. Merkel, J. Stoess, and F. Bellosa, Resource-conscious scheduling for energy efficiency on multicore processors. In *Eurosys*, 2010.

[30] PBZIP2. `http://compression.ca/pbzip2/`

[31] K. K. Pusukuri, D. Vengerov, A. Fedorova, and V .Kalogeraki. FACT: a framework for adaptive contention-aware thread migrations. In *CF*, 2011.

[32] K. K. Pusukuri, R. Gupta, L. N. Bhuyan. Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring. In IISWC, 2011.

[33] K. K. Pusukuri, R. Gupta, L. N. Bhuyan. No More Backstabbing... A Faithful Scheduling Policy for Multithreaded Programs. In *PACT*, 2011.

[34] K.K. Pusukuri and D. Johnson. Has one-thread-per-core binding model become obsolete for multithreaded programs running on multicore systems. In *USENIX HotPar*, 2013.

[35] K. K. Pusukuri, R. Gupta, L. N. Bhuyan. An Effective OS Load Balancing Technique for Multicore Multiprocessor Systems. Technical Report, Sept. 2012. University of California, Riverside.

[36] Z. Radovic and E. Hagersten. RH Lock: A Scalable Hierarchical Spin Lock. In *WMPI*, 2012.

[37] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura. Scalability-based manycore partitioning. In *PACT*, 2012.

[38] C. Severance and R. Enbody. Comparing gang scheduling with dynamic space sharing on symmetric multiprocessors using automatic self-allocating threads. In *IPPS*, 1997.

[39] A. Snavely, D.M. Tullsen, G. Voelker. Symbiotic Job scheduling For A Simultaneous Multithreading Processor. In *ASPLOS*, 2000.

[40] S. Sridharan, B. Keck, R. Murphy, S. Chandra, and P. Kogge. Thread migration to improve synchronization performance. In Workshop on Operating System Interference in High Performance Applications, 2006

[41] SPEC and the benchmark names SPEC OMP2001, SPEC jbb2005 are registered trademarks of the Standard Performance Evaluation Corporation. For more information, see `www.spec.org`.

[42] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. In *ISCA*, 1986.

[43] D. Tam, R. Azimi, and M. Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Eurosys*, 2007.

[44] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications.. In *ISCA*, 2011.

[45] L. Tang, J. Mars, and M. L. Soffa. Compiling For Niceness: Mitigating Contention for QOS in Warehouse Scale Computers. In *CGO*, 2012.

[46] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing GoogleâĂŹs Warehouse Scale Computers: The NUMA Experience. In *HPCA*, 2013.

[47] R. Thekkath and S. J. Eggers. Impact of Sharing-Based Thread Placement on Multithreaded Architectures. In *ISCA*, 1994.

[48] VMware ESX Server 2 NUMA Support. White paper. `http://www.vmware.com/pdf/esx2_NUMA.pdf`.

[49] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, 1995.

[50] F. Xian, W. Srisa-an, and H. Jiang. Contention-aware scheduler: unlocking execution parallelism in multithreaded java programs. In *OOPSLA*, 2008.

[51] X. Xiang, B. Bao, C. Ding, K. Shen: Cache Conscious Task Regrouping on Multicore Processors. In *CCGRID*, 2012.

[52] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *ASPLOS*, 2010.