Tumbler: An Effective Load Balancing Technique for MultiCPU Multicore Systems

KISHORE KUMAR PUSUKURI, University of California, Riverside
RAJIV GUPTA, University of California, Riverside
LAXMI N. BHUYAN, University of California, Riverside

Schedulers used by modern OSs (e.g., Oracle Solaris 11^{TM} and GNU/Linux) balance load by balancing the number of threads in runqueues of different cores. While this approach is effective for a single CPU multicore system, we show that it can lead to a significant load imbalance across CPUs of a multiCPU multicore system. Because different threads of a multithreaded application often exhibit different levels of CPU utilization, load cannot be measured in terms of the number of threads alone. We propose *Tumbler* that migrates the threads of a multithreaded program across multiple CPUs to balance the load across the CPUs. While Tumbler distributes the threads equally across the CPUs, its assignment of threads to CPUs is aimed at minimizing the variation in utilization of different CPUs to achieve load balance. We evaluated Tumbler using a wide variety of 35 multithreaded applications and our experimental results show that Tumbler outperforms both Oracle Solaris 11^{TM} and GNU/Linux.

Categories and Subject Descriptors: D.4.1 [**Process Management**]: Scheduling

General Terms: Scheduling, Operating Systems, Lock Contention

Additional Key Words and Phrases: Multicore, Load Balancing, Operating Systems

ACM Reference Format:
Pusukuri, K.K., Gupta, R., Bhuyan, L.N. 2015. Tumbler: An Effective Load Balancing Technique for MultiCPU Multicore Systems. ACM Trans. Arch. and Code Opt. V, N, Article A (January YYYY), 22 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Computing is progressing towards the use of machines that employ multiple multicore CPUs (or Sockets) to provide a large number of cores [Wentzlaff and Agarwal 2009]. While such systems present an opportunity to achieve high performance for multithreaded applications, achieving high performance on a multiCPU multicore system with a large number of cores is difficult due to the challenge of performing effective OS-level load balancing [Joao et al. 2012; Mendelson and Gabbay 2001; Peter et al. 2010].

Presence of load imbalance across multicore CPUs. Modern Operating Systems such as Oracle Solaris 11 and GNU/Linux employ dynamic load balancing techniques to achieve high system utilization on a multicore system. For balancing load, the OS approximates load by the number of threads in runqueues and migrates threads across cores to balance their runqueue lengths [McDougall and Mauro 2006]. This works fine for a single multicore CPU as long as cores are kept busy and, if a core runs out of threads, the OS migrates threads so all the cores can be kept busy. However, on a multiCPU multicore system, when different threads impose different load on the CPU, *the cumulative load of threads assigned to one CPU can vary significantly from the cumulative load of equal number of threads assigned to another CPU*. This can lead to performance degradation.

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

We have observed that, in practice, different threads impose different load on a CPU in two commonly arising scenarios: (i) uneven distribution of input load across identical threads or use of non-identical threads that perform different functions; and (ii) frequent synchronization or communication among threads. The OS thread scheduler, being unaware of the variation in load imposed by different threads, incorrectly assumes that balancing of runqueue lengths will also lead to balancing of load across multiple multicore CPUs. However, significant load imbalance across CPUs arises leading to variation in their utilizations. Here, *CPU utilization* is defined as the sum of the percentage of elapsed time a thread spends in *user space* and *kernel space* [McDougall and Mauro 2006].

Performance degradation due to load imbalance. The problem with load imbalance across CPUs, created by an uneven distribution of load across identical threads or the use of non-identical threads by the application, is that threads on a more heavily loaded CPU progress slowly. Thus, once the threads that are on a lightly loaded CPU finish their work, they may have to wait for other threads belonging to the same application that are still running on heavily loaded CPUs thus degrading overall performance. Let us consider a multithreaded application where large number of threads synchronize with each other frequently and thus create high lock contention. In this scenario, when a thread on a heavily loaded CPU acquires a lock, it may take longer to finish its critical section before releasing the lock. This further exacerbates the problem of lock contention and degrades performance. In other words, while load imbalance directly leads to increase in lock time, it can also indirectly cause thread latencies to increase and slow down the overall progress of the application. Since the OS load balancing algorithm does not consider whether the threads belong to the same application or not, thread migrations based on runqueue lengths for balancing load can be ineffective.

Here, the *lock time* is defined as the percentage of elapsed time a thread spends waiting for user-space locks and condition variables. The *thread latency* (or ready time) is defined as the percentage of elapsed time that a thread spends waiting for CPU resources. That is, although thread is ready to run, no CPU resources are available to schedule the thread [McDougall et al. 2006].

Tumbler -- Our Solution. To minimize load imbalance across multiple CPUs we must accomplish three things: (1) find the load imposed by each thread on the CPU; (2) migrate threads to balance the load; and (3) perform steps 1 & 2 efficiently so that the solution can scale to multiCPU multicore systems with a large number of cores. Our solution "Tumbler" achieves all of the above. Tumbler considers the collective CPU utilization by threads on a multicore CPU as the load they impose on that CPU. To balance load, Tumbler continuously monitors CPU utilization of individual threads and creates *thread groups* for each CPU such that the mean CPU utilization of each group is nearly the same. Since threads of high lock contention programs communicate often, their CPU utilization varies with time. Therefore for such applications Tumbler recomputes thread groups more frequently that is made practical by Tumbler's low runtime overhead. Tumbler employs a *variation-interval table* to select an appropriate *grouping interval* that represents how often thread groups are updated via inter-CPU thread migrations.

We evaluated Tumbler using 35 multithreaded applications including Data Caching benchmark from CloudSuite [Ferdman et al. 2012], SPECjbb2005 [SPECOMP 2001], PBZIP2 [pbzip2 2001], and programs from PARSEC [Bienia et al. 2008], SPEC OMP2012 [SPECOMP 2001], PHOENIX [Yoo et al. 2009], SPLASH2 [Woo et al. 1995] suites on a 64-core, 4-CPU machine. Tumbler outperforms both Solaris 11 and Linux. It achieves maximum of 37% performance improvement on Solaris (average 12%) and maximum of 26% on Linux (average 10%). It improves performance of Data Caching benchmark by up to 17%. Here, the performance improvement means reduction in the running time, except for Data Caching benchmark and SPECjbb2005 where improvement in throughput is reported. Moreover Tumbler *reduces performance variation* by up to 91% (average 42%). Tumbler also improves performance on *overloaded systems* (#threads > #cores) and is effective for *coscheduling* multiple multithreaded programs on multiCPU multicore systems. Tumbler outperforms the state-of-the-art cache contention management technique [Zhuravlev et al. 2010]. Finally, we have developed a portable implementation of Tumbler such that it does not require changes to the OS or the application code.

Related Work. Tumbler and Juggle [Hofmeyr et al. 2011] address entirely different causes of load imbalance. While Juggle focuses on balancing load *within* a multicore processor by restricting thread migrations to cores within the same processor, in contrast, Tumbler balances load *across* multiple multicore CPUs by migrating threads across CPUs. Juggle focuses on a oversubscribed system with *thread imbalance*, i.e. the number of threads is not a multiple of number of cores (e.g., running 11 threads on 8 cores). Instead, Tumbler focuses on dealing with *load imbalance across CPUs* because different threads impose different load on the CPU – this imbalance is present even when number of threads is a multiple of number of cores.

The *Thread Grouping* technique that is employed in Tumbler is different from the clustering technique proposed in [Tam et al. 2007]. Clustering techniques group similar objects to reduce variation within the clusters. In other words, they effectively maximize differences between clusters. However, Tumbler reduces the difference between means of thread groups. This is exactly opposite of thread clustering. To improve performance of multithreaded applications on multicore systems, thread migration has been considered before [Lozi et al. 2012; Sridharan et al. 2006]; however, these solutions require modification of either the application [Lozi et al. 2012] or the OS [Sridharan et al. 2006]. The solution we develop can be applied to any application on-the-fly with no need to modify the application or the OS. In contrast, several works [Zhang et al. 2010] employ one-thread-per-core *binding* model to maximize performance of multithreaded programs on multicores. While this works for multicore systems with a small number of cores (4 or 8 cores), it does not work for systems with a large number of cores, specifically for multithreaded programs that involve high lock contention [Pusukuri et al. 2014].

The key contributions of our work are as follows:

— We demonstrate that modern OSs experience load imbalance across multiple CPUs leading to significant performance degradation for multithreaded applications running on multiCPU multicore systems.
— We develop the Tumbler load balancing technique that greatly improves performance of a wide variety of multithreaded applications running on a multiCPU multicore system with a large number of cores.
— We demonstrate that Tumbler not only outperforms Linux and Solaris, it also outperforms the DINO [Zhuravlev et al. 2010] state-of-the-art cache contention management technique for multiCPU systems.
— We show that overhead of Tumbler is negligible making it scalable; it is also portable across OSs as it does not require changes to the OS or the application code.

The remainder of this paper is organized as follows. Section 2 demonstrates that modern OS load balancing algorithms lead to load imbalance across multiple sockets of a multicore system. Section 3 presents our solution, Tumbler, and Section 4 describes its implementation. In Section 5, we evaluate Tumbler. Related work and conclusions are given in Sections 6 and 7.

2. MOTIVATION: DEMONSTRATING LOAD IMBALANCE ACROSS CPUS

Modern OSs such as Solaris and Linux migrate threads across cores of a multicore system to balance load. The load is approximated by the number of threads in the runqueues. However, number of threads alone is an inadequate measure of load for multithreaded programs running on a multicore multiCPU system. This is because of the two commonly observed situations: *uneven distribution of load across threads* causing them to impose different load on CPUs; and *frequent synchronization and communication* among threads causing their CPU utilization to vary over time. In this section we present motivating experiments illustrating that the above situations can lead to significant performance load imbalance across multiple CPUs. For illustration we use programs from the PARSEC [Bienia et al. 2008] suite and observe the load imbalance they create when run on our machine shown in Figure 1 that provides 64-cores distributed across four CPUs.

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);

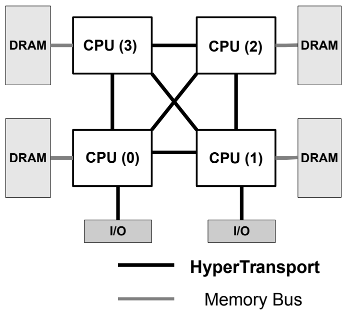


Fig. 1: Our 64-core machine has four 16-core CPUs. We interchangeably use CPU and Socket.

Table I: The mean CPU utilization (%) of threads in the intermediate pipeline stages of *Ferret*.

Segment	Extract	Vector	Rank
2.0 %	1.0%	11.0%	36%

Uneven distribution of load across threads. is due to two reasons. In some applications multiple threads of different kind may be required to perform different tasks on different data causing them to exhibit different CPU utilization. Even when all threads are identical, the input load may have been distributed unevenly across the different threads causing them to impose different CPU utilization.

Consider the situation of a program that makes use of pipelined parallelism. A pipelined workload breaks the computation into *pipeline stages* and executes them concurrently on a multicore system. Each stage typically has multiple threads assigned to it to maximize its throughput [Bienia et al. 2008]. Typically, the worker threads in different stages carry different work load and thus they exhibit different levels of CPU utilization. For example, the ferret (FR) program from the PARSEC suite is divided into six pipeline stages – the results of processing in one stage are passed on to the next stage. The stages are: Load, Segment, Extract, Vector, Rank, and Out. The first and last stage have a single thread and each of the intermediate stages have a pool of *n* threads. Table I shows the mean CPU utilization of the threads in the intermediate pipeline stages of ferret. As we can see in Table I, CPU utilization of threads in different pipeline stages varies significantly and thus placing equal number of threads across sockets can lead to significant variation in the CPU utilization across multiple Sockets.

When we run FR with schedulers on Linux and Solaris, and then with Tumbler, FR is observed to achieve maximum performance with Tumbler. As Table II shows, the variation in the CPU utilization of the four CPUs is high for both Linux (17% to 34%) and Solaris (18% to 35%) while it is quite low for Tumbler (23% to 26%). The variation in CPU utilization across the four CPUs with both Linux and Solaris is almost the same. Note that both OS schedulers consider number of threads as the load and migrate threads across to cores to balance the runqueue lengths. However, Tumbler considers CPU utilization of individual threads and divides them into groups of equal number of threads that collectively place nearly the same load on each CPU. Tumbler reduces variation in CPU utilization across the four CPUs by around 80% compared to Linux and Solaris. It improves the performance by roughly 11%.

Frequent synchronization or communication. Frequent interaction among multiple threads of an application causes thread CPU utilization to vary significantly over time, i.e. CPU utilization is low when a thread is waiting as opposed to when it is running. Consider Streamcluster (SC), whose implementation is based upon pthreads, that solves the online clustering problem. SC finds a predetermined number of medians so that each input point can be assigned to its nearest center. SC continuously organizes data produced under real-time conditions (e.g., for network intrusion

Table II: CPU utilization (%) of the four Sockets for *Ferret* showing variation in CPU utilization resulting from uneven distribution of load across threads in different pipelined stages.

Algorithm	CPU(0)	CPU(1)	CPU(2)	CPU(3)
Linux	17%	23%	26%	34%
Solaris	18%	21%	26%	35%
Tumbler	23%	25%	26%	26%

Table III: Frequent synchronization: CPU utilization (%) of Sockets for *SC* with load balanced across 64 threads.

Algorithm	CPU(0)	CPU(1)	CPU(2)	CPU(3)
Linux	17%	23%	28%	32%
Solaris	16%	27%	28%	29%
Tumbler	20%	26%	27%	27%

detection). SC spends most of its time evaluating the gain of opening a new center. This operation is based upon a parallelization scheme which employs *static* partitioning of data points [Bienia et al. 2008]. On the default workload of one million data points (1,000,000), we ran SC with 64 threads on our 64-core machine with four CPUs. Note that input of 1,000,000 points is evenly distributed across 64 threads.

Table III shows that the CPU utilization varies from a minimum of 17% (16%) to a maximum of 32% (29%) for Linux (Solaris). In contrast load imbalance across the four sockets is lower for Tumbler. This is because even though threads are identical, their CPU utilization varies over time – this can be observed from Figure 2 which shows the time varying CPU utilization of two threads chosen randomly from among the 64 threads being run.

Let us discuss the reasons for the variation in CPU utilization of a given thread. In the presence of high lock contention, the CPU utilization of threads varies based upon the current state of the thread. A thread can mainly be in three different *r*esource usage states: (i) utilizing CPU resources for computation; (ii) utilizing CPU resources for spinning; and (iii) sleeping -- e.g., waiting for locks (contributing to lock times and lock acquisition latencies). This is because of how the locks are implemented – on Solaris, the pthread adaptive mutex uses the spin-then-block lock contention management policy. It is based on the assumption that mutex hold times are typically short enough that the time spent spinning is less than the time it takes to block. As a result, threads typically spin if the lock-holder thread is running on another core and block otherwise [McDougall and Mauro 2006; Johnson et al. 2010]. In user-space, a thread spins 1000 times while trying to grab the lock. If the spinning fails, then libc will park (reschedule and block) the thread via the lwp_park() interface [McDougall and Mauro 2006].

Let us analyze the impact of interaction between OS level load balancing and the communication across threads in SC. The OS thread scheduler migrates threads not only for load balancing, it also migrates them to ensure that all threads make progress. For example, when a thread transitions from sleep state to a ready-to-run state (e.g., due to a lock operation), if the core on which it last ran is not available, then it is likely to be migrated to another available core [McDougall and Mauro 2006]. These thread migrations often disturb the load balance across CPUs. We observed that high lock contention programs experience high thread migration rates compared to low lock contention programs. For example, due to high lock contention streamcluster (SC) experiences 1170 thread migrations per second across the 64 cores while a lock free program swaptions (SW), also from the PARSEC suite, experiences 10 thread migrations per second.

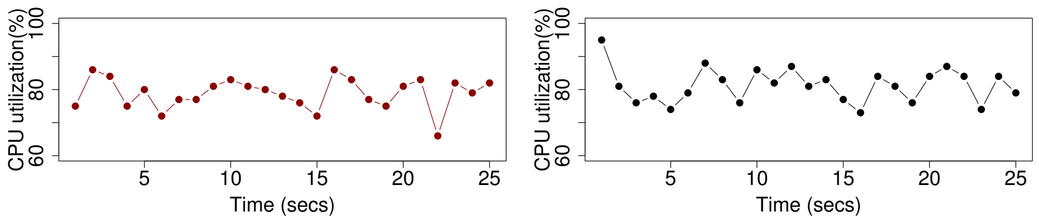


Fig. 2: CPU utilization of two threads of SC.

Finally, communication between threads due to synchronization causes increases in lock times to spread across threads of a multithreaded application. This is because some threads may need to wait longer for acquiring locks released by other threads. This leads to high variation in CPU utilization across threads and increased LLC (last-level cache) misses due to inter-CPU lock transfers that fall on the critical path of execution. The latter causes increases in lock acquisition latencies of threads. The increase in lock times and lock acquisition latencies affect the CPU utilization of threads.

A common synchronization primitive that causes load imbalance is mutex. However, there are other synchronization primitives that also cause load imbalance. For example imbalanced load distribution combined with barrier in fluidanimate from PARSEC; imbalnced load distribution combined with condition variables in facesim from PARSEC; and read/write locks in bodytrack from PARSEC.

By considering CPU utilization of threads as load imposed by threads, Tumbler migrates threads across CPUs to maintain balanced load across the CPUs. This naturally leads to reduced lock times, lock acquisition latencies, and LLC misses. While the SC benchmark exhibits lock time of 21.4% with default Solaris, with Tumbler the lock time is reduced to 7.9%. Furthermore, the LLC miss rate (misses per thousand instructions) is reduced from 11.3 to 8.7 by Tumbler. Tumbler reduces variation in CPU utilization across CPUs by 63% and improves the overall performance of SC by 28%. The variation is expressed as the coefficient of variation (CV) – the ratio of standard deviation to mean.

3. TUMBLER

We identified two situations that lead to load imbalance across Sockets/CPUs. When load is distributed unevenly across threads, the time the threads spend in *user space* varies across threads. When we are faced with high contention programs, the times threads spend in *kernel space* varies across threads. Thus, a good way to estimate the load that a thread places on the CPU is to measure the time it spends both in user space and kernel space. Given a thread with id, *tid*, and a time interval Δ, we can express the average load that the thread places on the CPU over the time interval Δ, denoted as $CPU(tid,\Delta)$, as follows:

$$CPU(tid,\Delta) = \frac{U(tid,\Delta)+K(tid,\Delta)}{\Delta} \tag{1}$$

where $U(tid,\Delta)$ and $K(tid,\Delta)$ are the times *tid* spent in user space and kernel space respectively. The goal of Tumbler is to balance load across the sockets by minimizing the difference between the cumulative loads imposed by thread groups TG_i and TG_j assigned to every pair of Sockets CPU_i and CPU_j (shown in Equation 2).

$$\left| \sum_{\forall tid \in TG_i} CPU_i(tid,\Delta) - \sum_{\forall tid \in TG_j} CPU_j(tid,\Delta) \right| \tag{2}$$

On Solaris, we are able to *p*recisely measure CPU utilization which is defined as the sum of the percentage of elapsed time thread spends in user space and kernel space. The Solaris proc filesystem provides fine grain details of resource usage by threads -- how much time threads spend in user space

Algorithm 1: The Tumbler.

```
Input: N: Number of threads; P: Number of CPUs
1 repeat
2     Monitor Threads -- sample CPU utilization values of N threads;
3     Identify Thread Groups -- divide threads into P groups via Thread Grouping algorithm –
      Algorithm 2;
4     Affect Thread Grouping -- assign one multicore CPU to each one of the groups;
5     Select Grouping Interval -- select grouping interval using Variation-interval Table;
6 until program terminates;
```

lock operations (i.e., lock time), ready queues (thread latency), etc [McDougall and Mauro 2006; McDougall et al. 2006]. Thus, we are able to precisely measure the percentage of time a thread utilizes CPU resources in user space and kernel space. While uneven distribution of load reflects in thread latency data, lock contention (lock transfers) is reflected in lock time data. We are able to exclude lock times and latency times and precisely derive CPU utilization of threads in Solaris. We consider CPU utilization of a thread as being the measure of the load imposed by the thread.

Next we present the detailed design of Tumbler that balances load across multiple CPUs by continuously monitoring CPU utilization of threads and creating *thread groups* for each CPU such that the mean CPU utilization by threads in each group is nearly the same. In high lock contention programs, threads communicate often and thus their CPU utilization also varies with time; therefore Tumbler recomputes thread groups more frequently. Tumbler employs a *variation-interval table* for selecting the grouping interval that represents how often thread groups are updated by migrating a subset of threads across CPUs.

The benefits of employing Tumbler include the following. Improvements in load balancing lead to reduction in lock times and thus improved performance for multithreaded programs. For high lock contention programs, lock acquisition latencies and LLC misses on the critical path are reduced. Therefore, Tumbler speeds up the execution of shared data accesses in critical section. Moreover, Tumbler does not require any changes to the application source code or the OS kernel and can be applied on-the-fly to any application that is run on the system.

The overview of Tumbler is provided in Algorithm 1. Tumbler is implemented by a daemon thread which executes throughout an application's lifetime repeatedly performing the following four steps: *monitor* CPU utilization values of threads; *identify* thread groups; *affect* thread grouping; and *select* grouping interval. The first step monitors the CPU utilization values of threads at regular intervals. The second step groups threads using Algorithm 2 to reduce the variation in *mean* CPU utilization of different groups. The third step simply affects the thread groups via thread migrations. Finally, the fourth step selects an appropriate grouping interval using *variation-interval table*. The details of these steps follow.

3.1. Monitoring Threads

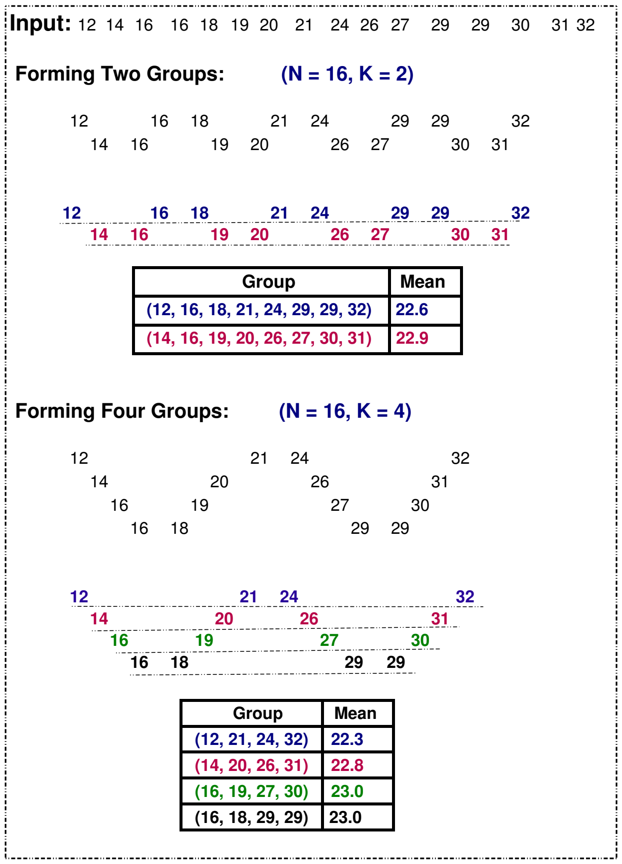
Tumbler considers CPU utilization of threads for balancing load. It continuously monitors the CPU utilization values of threads through the proc file system in regular monitoring intervals. Tumbler maintains per thread profile data structure that holds the CPU utilization (%) values collected for the thread as well as the id of the thread. We use grouping interval also as monitoring interval -- we read the CPU utilization values of threads at regular grouping intervals through the proc file system. After getting the CPU utilization values of threads, we perform other steps as shown in Algorithm 1.

3.2. Thread Grouping: Identifying Thread Groups

At regular intervals (i.e., *grouping interval*), Tumbler examines the CPU utilization profile data collected for the threads, and constructs one thread group per CPU. Tumbler uses the collective CPU utilization of threads on a multicore CPU as their load and reduces the variation in mean CPU utilization of different groups. Our thread grouping technique sorts the threads according to their

Algorithm 2: Thread Grouping: produces K groups of N/K threads using N Thread profile structures such that the difference between means of CPU utilization values of the groups is minimized.

```
   Input: T[N]: N Thread profile structures.
   Output: K groups of threads (K == #CPUs).
1  Sort T[] based on their CPU utilization values;
2  for i = 0 to (N - 1) do
3      if ((i / K) % 2) == 0 then
4          group[i % K] ⟵ T[i];
5      else
6          group[K - (i % K) - 1] ⟵ T[i];
7      end
8  end
9  return groups;
```



Group	Mean
(12, 16, 18, 21, 24, 29, 29, 32)	22.6
(14, 16, 19, 20, 26, 27, 30, 31)	22.9

Group	Mean
(12, 21, 24, 32)	22.3
(14, 20, 26, 31)	22.8
(16, 19, 27, 30)	23.0
(16, 18, 29, 29)	23.0

Fig. 3: Working of the thread grouping technique. It scans the sorted input as a *wave form*, and produces groups of threads of the corresponding CPU utilization values.

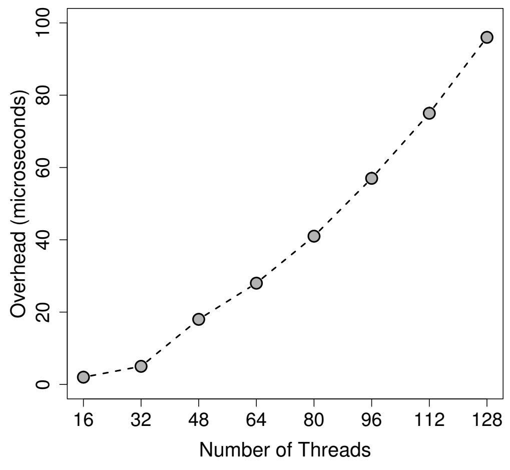


Fig. 4: Overhead of identifying groups of threads.

CPU utilization values and then divides them into as many groups of consecutive threads as the number of CPUs being used to run the application. For example, on our system, if all CPUs are being used, the first group is assigned to CPU(0), the second to CPU(1), the third to CPU(2), and the fourth to CPU(3). We maintain one profile data structure per thread. The thread profile data structure includes thread id, CPU utilization (%), and group id of the thread. Moreover, we continuously monitor variation in CPU utilization of threads.

Algorithm 2 shows our Thread Grouping technique and Figure 3 illustrates its working. To simplify the illustration of the Thread Grouping technique we consider 16 data points corresponding to CPU utilization (%) values of 16 threads. As we can see in Figure 3, the algorithm scans the sorted input as a *wave form*, and produces groups of threads of the corresponding CPU utilization values to reduce the variation in mean CPU utilization of different groups. The resulting groups and their mean CPU utilization values are also shown in Figure 3.

Overhead of Thread Grouping.. For 64 thread profiles (i.e., 100% load), grouping takes an average of 28 microseconds on our machine. Therefore its overhead is negligible. The time complexity of the grouping algorithm is O(N log(N)). Although we run programs with #threads == #cores configuration in this work, our grouping technique also works well for #threads > #cores configuration. For example, as Figure 4 shows, for set of 128 thread profiles (i.e., 200% load), it takes an average of 94 microseconds on our machine.

3.3. Affecting Thread Grouping

Next, the thread groups computed in the preceding step will be assigned to CPUs. At a regular time interval, called the *grouping interval*, Tumbler simultaneously migrates as many threads as needed to realize the new thread groups computed in the preceding step. The pset_bind_lwp(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 [McDougall and Mauro 2006]. Tumbler migrates threads across CPUs (through regrouping threads in regular intervals) for balancing load across CPUs and delegates the task of balancing load *within cores of a CPU* to the default Solaris thread scheduler. However, typically only a subset of threads need to be migrated across CPUs because many threads are already bound to the set of cores on the CPUs where we want them to be.

Table IV: The variation-interval table.

Variation in CPU Utilization (CV)	**Grouping Interval (millisecs)**
($\geq$ 0.40)	50
(0.40 -- 0.36)	100
(0.35 -- 0.31)	150
(0.30 -- 0.21)	200
(0.20 -- 0.11)	400
(0.10 -- 0.05)	800
($<$ 0.05)	*No Grouping*

3.4. Selecting Grouping Interval

Since threads of high lock contention programs communicate with each other very often due to synchronization (e.g., locks), we should use appropriate grouping interval to decide how often threads should be grouped and migrated across CPUs. Threads of high lock contention programs exhibit high variation in their CPU utilization compared to threads of low lock contention programs. Therefore, based on the variation in CPU utilization of threads we need to choose appropriate grouping interval.

Grouping interval also impacts the performance improvements as it affects data locality, LLC misses, and consequently the overall CPU utilization of the threads. When programs exhibit phase changes, there is often high variation in their resource usage. Therefore, we may need to continuously select appropriate grouping interval to effectively deal with phase changes. Intuitively the programs that exhibit high variation in the CPU utilization by their threads need shorter grouping interval. Therefore, for *dynamically* selecting an appropriate grouping interval, we make use of a *variation-interval table*.

For developing the variation interval table, we categorize the programs used in this work as memory intensive and CPU intensive. If the LLC miss rate (MPKI) of a program is greater than 3 then we consider the program to be memory intensive; otherwise it is considered to be CPU intensive. We selected a few applications from each category, a total of 8 out of 35 applications used in our experiments, and ran them with varying grouping intervals from 20 milliseconds to 1000 milliseconds. We observed the effect of different grouping intervals on the performance of programs and also how the coefficient of variation for CPU utilization of the threads of programs varied with different grouping intervals. Based upon these observations, we developed a variation-interval table shown in Table IV.

As we can see from Table IV, the grouping interval goes up as the coefficient of variation (CV) of the CPU utilization of programs goes down. The 8 applications used to populate the variation interval table are: bodytrack, fluidanimate, facesim, streamcluster, swaptions, swim, applu, and mgrid. The variation-interval table obtained was then used in our experiments for all 35 applications.

Thus, Tumbler dynamically applies appropriate grouping interval using the Variation-interval Table to deal with phase changes as well as to reduce variation in CPU utilization of threads. It is also possible that in some programs, over many intervals, the CPU utilizations of threads may 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 migration step will be skipped and monitoring will be resumed. In other words, when thread migrations are not expected to yield any benefit, they will not be performed.

Grouping Interval and Memory-intensive Programs. Tumbler performs thread migration at regular relatively long intervals compared to the default OS scheduler. Thus, for high lock-contention and memory-intensive programs, the benefits of load balance outweigh temporary LLC misses caused by thread migrations. Moreover, the overall impact on LLC miss rate is small because, by balancing load, Tumbler significantly reduces lock transfers (and consequently LLC misses) between sockets.

Grouping Interval and Multiple Applications. Although, we selected the group interval by running one application at a time on the system, we see that it is also works well while running multiple applications as Tumbler considers CPU utilization values of all the threads (belong to multiple applications) in grouping threads and assigning them to processor-sets.

In summary, Tumbler continuously monitors CPU utilization of threads in a program and effectively groups them to reduce the variation in mean CPU utilization of different groups. Consequently, it reduces lock time and also reduces both LLC miss rates and thread latencies, resulting in improved performance. *We have developed a portable implementation of Tumbler which does not require changes to the OS or the application. Therefore, it is easily portable to any OS where access to CPU utilization of individual threads within application is available.*

4. IMPLEMENTATION

Tumbler is implemented as a daemon thread in user space, because, our goal is to develop a solution that can be easily applied without modifying the application or the OS. It may be possible to modify application code to alleviate the load imbalance problems described in this work. However, in general, this may not be an easy task as the behavior of the program may be input sensitive. Also the general solution we have developed may be implemented in the OS. However, we have followed an approach that makes Tumbler easy to implement and apply to applications by leveraging the advanced utilities supported by modern OSs. This approach gives us portability and in fact we were able to apply this strategy for rapid prototyping our solution for both Solaris and Linux. Thus, our approach is simpler than either changing the OS kernel or modifying all the applications. However, Tumbler needs root privileges for creating processor-sets (a pool of cores) as it also deals with multiple applications.

Roles of Tumbler and OS Scheduling Policy. Scheduling policies are orthogonal to load balancing techniques: while a scheduling policy assigns priorities and time quanta to threads to provide fairness among threads, a load balancing technique migrates threads between CPUs to balance load and improve overall system utilization. Tumbler is a load balancing technique and it does not affect OS scheduling policies. The main task of Tumbler is to migrate threads between Sockets for minimizing variation in CPU utilization of threads. In other words, Tumbler simply affects which threads are located on each of the Sockets. Moreover, Tumbler's approach is consistent with default OS load balancer, as both uniformly distribute the threads across Sockets. Thread migrations among the cores on each Socket are done by the default OS load balancer and when to run which thread and for how long is decided by the default OS scheduling policy.

5. EVALUATION

We demonstrate the merits of Tumbler via experimental evaluation of the performance of several multithreaded applications on both Oracle Solaris 11 and Linux (Ubuntu 12.04 LTS, kernel 3.2.0). We also provide a detailed explanation of the overheads associated with Tumbler. We seek answers of the following questions through these experiments:

— Does Tumbler achieve better performance for multithreaded programs than Solaris and Linux?
— How does Tumbler perform compared to the state-of-the art cache contention management technque DINO [Blagodurov et al. 2011] and PBind (pinning one thread to core)?
— How does Tumbler perform on overloaded systems (#threads > #cores)?
— Is Tumbler effective in running multiple multithreaded programs simultaneously?

5.1. Experimental Setup

Our experimental setup consists of a 4-CPU 64-core machine. We evaluate Tumbler on both Oracle Solaris 11 and GNU/Linux (Ubuntu 12.04.4 LTS, kernel 3.2.0). Figure 1 shows the machine configuration.

Benchmarks. To evaluate Tumbler we considered 44 multithreaded programs including Data Caching Benchmark [Ferdman et al. 2012; Fitzpatrick 2004], SPECjbb2005 [SPECOMP 2001], PBZIP2 [pbzip2

Table V: Programs and their short names.

Data Caching Benchmark (from CloudSuite); **PARSEC:** bodytrack (BT), canneal (CA), fluidanimate (FA), freqmine (FQ), ferret (FR), facesim (FS), streamcluster (SC), swaptions (SW); **SPEC OMP2012:** applu331 (AL), botsalgn (BA), bt331 (BB), botsspar (BS), bwaves (BW), fma3d (FM), ilbdc (IL), imagick (IM), kdtree (KD), md (MD), mgrid331 (MG), nab (NB), smithwa (ST), swim (SM); **SPLASH2:** lu (LU), ocean (OC), raytrace (RT), radix (RX), volrend (VL), water (WA); **PHOENIX:** kmeans (KM), pca (PC), string match (SR), word count (WC); **SPEC jbb2005** (JB); **PBZIP2** (PB).

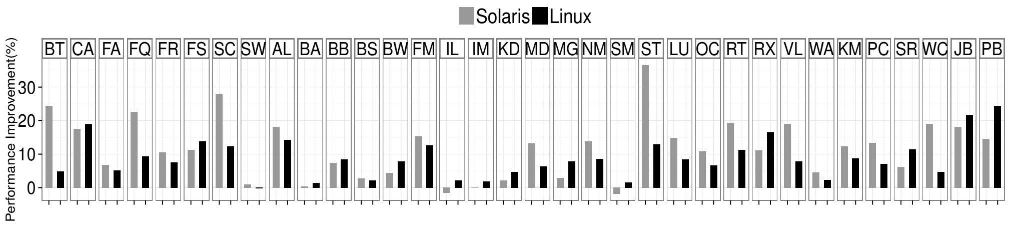


Fig. 5: Tumbler improves performance of a wide variety of programs on both Solaris and Linux. It achieves maximum of 37% performance improvement on Solaris and 24% on Linux.

2001], and programs from PARSEC [Bienia et al. 2008], SPEC OMP2012 [SPECOMP 2001], PHOENIX [Yoo et al. 2009], and SPLASH2 [Woo et al. 1995] suites. The implementations of the PARSEC, PHOENIX, and SPLASH2 are based upon *pthreads* and we ran them using the largest inputs available. SPEC OMP2012 programs were run on medium sized inputs. SPEC jbb2005 with single JVM is used. We present detailed performance data for the 35 programs of the above 44 programs as they have substantial parallelism and have long running times ($\geq$ 10 seconds). Table V lists the 35 programs and their short names. We exclude the remaining nine programs because either they have very short running times ($<$ 10 seconds) or they do not have enough parallelism. The nine programs are: blacksholes, dedup, x264, vips from PARSEC; cholesky, barnes, fft, fmt from SPLASH2; and histogram from PHOENIX.

Performance Metrics. We ran each experiment 10 times and present average (the arithmetic mean) and coefficient of variation (CV) for the 10 runs. The performance metrics we use are *percentage reduction in running time* for all the programs except for Data Caching Benchmark and SPECjbb2005 where *improvement in throughput* is presented.

We evaluate Tumbler with the Data Caching benchmark by varying number of client threads: 32, 64, 96, and 128. We ran the remaining programs with 64 threads in all the experiments except in the overloaded system experiments (Section 5.5) and coscheduling experiments (Section 5.4). Section 5.3 explains the performance improvements of Data Caching Benchmark with Tumbler.

5.2. Performance Benefits

Figure 5 shows that Tumbler improves performance of a wide variety of programs, listed in Table V, on both Solaris and Linux. It achieves maximum of 37% performance improvement on Solaris (average 12%) and maximum of 26% on Linux (average 10%). The performance improvements via Tumbler on Solaris and Linux differ due to differences in the two OSs. On Solaris, we are able to precisely measure CPU utilization (user space + kernel space) of threads. Solaris proc filesystem provides fine grain details of resource usage by threads -- how much time threads spend in user space lock operations (i.e., lock time), ready queues (thread latency), etc [McDougall and Mauro 2006]. Thus, we are able to precisely measure the percentage of time a thread utilizes CPU resources in user space and kernel space. We are able to exclude lock times and latency times and precisely derive CPU utilization of threads in Solaris. However, on Linux we *estimate* CPU utilization of threads by collecting user and kernel time from the task stat file (/proc/pid/task/tid/stat) which does not provide lock times and latency times. This is one of the important reasons why Tumbler achieves lower performance improvements on Linux than on Solaris.

Next, we present the performance data that shows why Tumbler is superior than the existing state-of-the-art techniques. Since Solaris provides several effective low-overhead observability tools (e.g., DTrace [Cantrill et al. 2004]). We did this performance analysis on Solaris. On Solaris, we compare Tumbler with the following:

(i) DINO. [Blagodurov et al. 2011] is a state-of-the-art cache contention management technique. It reduces cache contention (LLC miss rate) by separating memory intensive threads by scheduling

Table VI: Performance improvements **relative to Solaris**. Tumbler improves performance by up to 37% and average of 12%. Tumbler outperforms DINO, PSets, and Pbind.

	Performance Improvement (%)			
Program	**Tumbler**	**DINO**	**PSets**	**PBind**
BT	**24.3**	-5.0	-1.9	-3.1
CA	**17.6**	1.3	3.2	-5.2
FA	**6.8**	2.1	7.0	-1.7
FQ	**22.7**	3.5	1.4	-10.4
FR	**10.5**	0.8	-2.1	-23.6
FS	**11.3**	-13.2	1.1	1.0
SC	**27.8**	6.0	-13.0	-15.0
SW	**1.0**	-1.8	0.4	-1.2
AL	**18.1**	6.2	4.1	-9.4
BA	**0.3**	-1.7	0.8	0.1
BB	**7.4**	5.1	2.7	2.5
BS	**2.7**	-0.1	1.2	-5.8
BW	**4.4**	6.9	0.8	-3.6
FM	**15.3**	-4.5	-1.3	-19.5
IL	**-1.5**	-1.3	1.9	2.4
IM	**-0.1**	-1.9	0.4	-0.7
KD	**2.2**	-1.1	2.5	0.4
MD	**13.2**	-7.2	2.1	0.6
MG	**2.9**	-0.8	0.6	-6.0
NM	**13.8**	-8.1	1.4	-3.6
SM	**-1.9**	-2.6	1.2	-6.2
ST	**36.6**	-3.1	6.5	1.1
LU	**14.8**	9.4	2.8	-11.2
OC	**10.9**	3.1	4.2	2.1
RT	**19.2**	-10.9	2.0	-4.0
RX	**11.1**	5.0	-13.0	-7.4
VL	**19.0**	-7.0	2.8	-1.1
WA	**4.5**	-1.5	3.1	-2.0
KM	**12.3**	8.5	5.7	1.9
PC	**13.3**	2.1	2.0	-4.0
SR	**6.2**	-11.4	-4.5	-9.2
WC	**19.0**	5.2	2.8	3.1
JB	**18.1**	-2.0	-0.9	-7.4
PB	**14.5**	11.2	-12.9	-21.0

them on different CPUs of a multicore machine. By combining low memory-intensive threads with high memory-intensive threads, the cache pressure is balanced across CPUs.

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

(iii) 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 CPUs)* with PSets.

Table VI shows that Tumbler significantly outperforms default Solaris and all other techniques. Substantial performance improvements (12% on average) are observed over the default Solaris -- more than 10% for 21 programs with a maximum of 37% improvement, 4%-10% for five programs, and less than 4% for five programs. The throughput of SPEC jbb2005 (JB) is improved by 18%. The programs exhibiting high lock contention yield high performance improvements with Tumbler. Figure 6 shows that Tumbler reduces variation in CPU utilization of threads, lock times, LLC miss rates, and thread latencies compared to the default Solaris.

Moreover, Tumbler not only improves performance and also reduces system noise. That is why Tumbler significantly reduces performance variation (coefficient of variation in running times). As Figure 7 shows, Tumbler reduces performance variation up to 91% and an average of 42%. Minimizing performance variation or improving performance predictability is critical for effective resource management and performance regression analysis [Curtsinger and Berger 2013; Pusukuri et al. 2014].

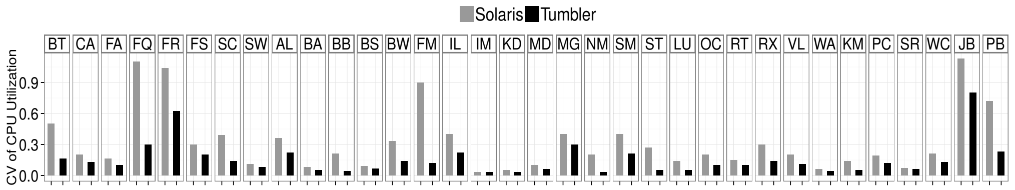
On Solaris, Tumbler slightly degrades performance of two programs (IL, SM) and there is no improvement for IM. This is because: (i) the variation in CPU utilization of threads of IM is very low and therefore Tumbler does not migrate threads; (ii) the programs (IL, SM) are extremely memory intensive and also have very large working sets ($>$ 12 GB). Both IL and SM experience high LLC miss rates with Tumbler compared to Solaris. However, although Tumbler slightly increases LLC miss rate of MG, overall MG achieves high performance as its thread latencies and lock times are reduced by Tumbler. The damage caused data locality of programs (e.g., IL), by thread migration, outweighs the benefit from reduced thread latency and lock time.

PSets performs slightly better than DINO. DINO outperforms Solaris for memory intensive programs with low lock contention programs -- PB, FA, RX, KM and SC. However, the default Solaris scheduler is better *on average* compared to DINO. 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 does not work well for multithreaded programs with *high lock contention*. One can view Tumbler and DINO as complementary techniques – DINO can be used for lock contention free programs and memory intensive programs that exhibit low thread latencies.

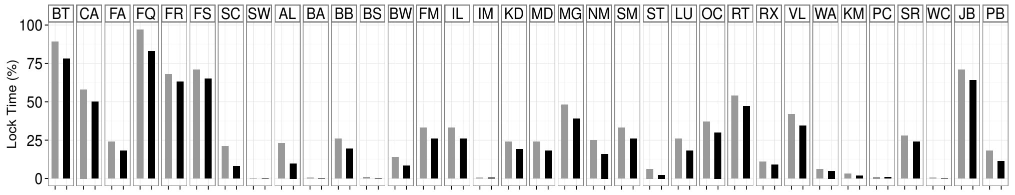
PBind significantly degrades performance of several programs. By restricting thread migrations across cores, PBind increases the severity of load imbalance across CPUs. PBind also gives poor performance for programs that involve lock contention on multiCPU multicore systems with a large number of cores. As the threads of a multithreaded program do not migrate across CPUs of a multicore system with PBind, the frequency of high overhead lock transfers between CPUs increases compared to the default Solaris. This leads to an increase of LLC misses in the critical path of the multithreaded program, which leads to poor performance [Pusukuri et al. 2014].

5.3. Data Caching Benchmark with Tumbler

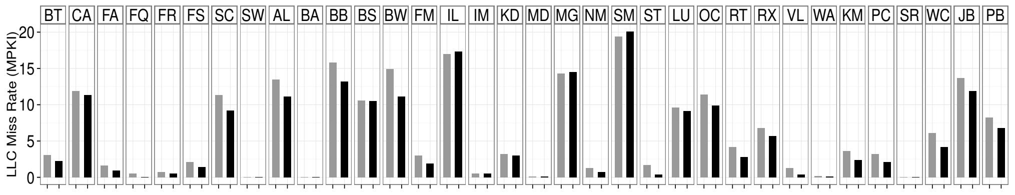
As Cloud applications are becoming data-intensive and spend most of their execution time waiting on memory, it is very important that data is available within hundreds of microseconds. That is why most



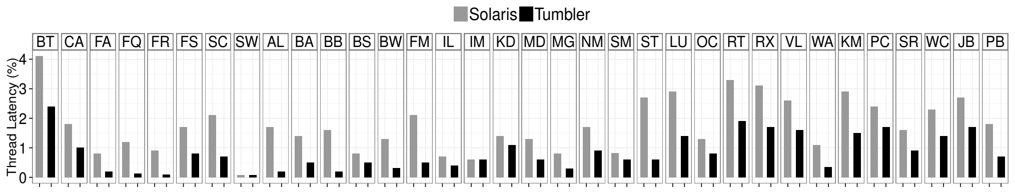
(a) Coefficient of variation of CPU utilization of threads.



(b) Lock Time.



(c) LLC miss rate.



(d) Thread Latency.

Fig. 6: Tumbler reduces variation in CPU utilization of threads. Consequently it reduces lock times, LLC miss rates, and thread latencies compared to the default Solaris.

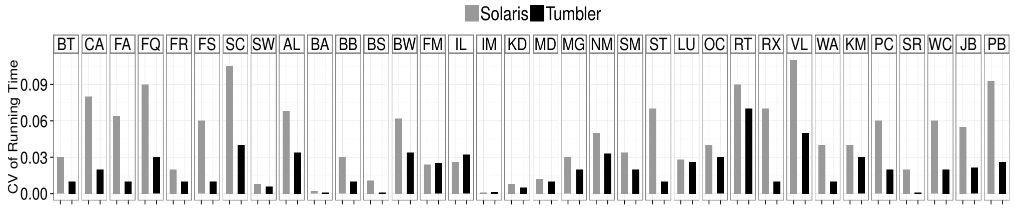
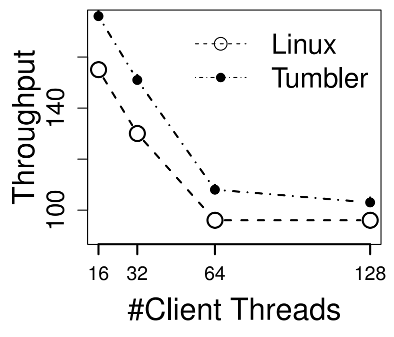
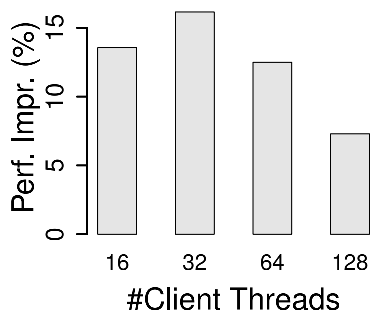


Fig. 7: Tumbler reduces coefficient of variation of running times (i.e., performance variation).

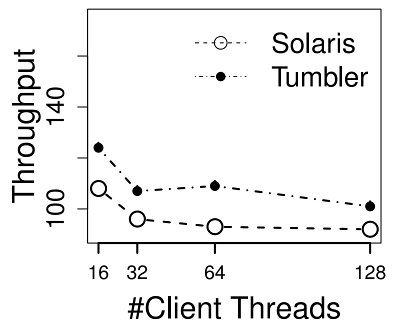


(a) Throughput.

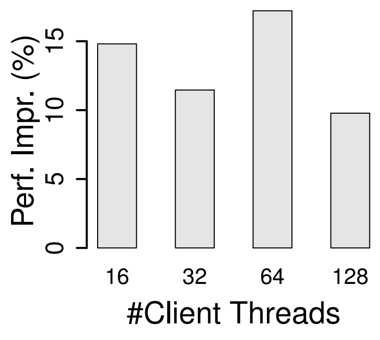


(b) Performance improvement (%).

Fig. 8: **Linux vs Tumbler**: Throughput is improved by Tumbler - one memcached server is configured with 4 threads and stressed varying number of client threads.



(a) Throughput.



(b) Performance improvement (%).

Fig. 9: **Solaris vs Tumbler**: Throughput is improved by Tumbler - one memcached server is configured with 4 threads and stressed varying number of client threads.

of todays server systems use dedicated caching servers (e.g., memcached) that cache the data in their DRAM instead of using hard disks. In-memory database servers are also use dedicate caching servers. Therefore we evaluate Tumbler with the Data Caching benchmark from CloudSuite1.0 [Ferdman et al. 2012; Fitzpatrick 2004]. The Data Caching benchmark relies on the most widely used data-caching platform, memcached, and simulates a Twitter caching server using a real Twitter dataset.

We ran one memcached server with 4 threads as it scales poorly beyond four threads [Ferdman et al. 2012; datacache 2012]. We stressed the memcached server threads with the Data Caching benchmark by varying number of client threads (32 -- 50%load , 64 -- 100% load, 96 -- 150% load, and 128 -- 200% load) on our 64-core machine. Figures 8(a) and 9(a) show the throughput (kilo requests per second) achieved by the default Linux, default Solaris, and Tumbler. As we can see, Tumbler delivers higher throughput across different number of client threads.

Figures 8(b) and 9(b) show Tumbler improves performance of Memcached by up to 15% compared to Linux and up to 17% compared to Solaris. Tumbler not only improves throughput, it also reduces latency by up to 11% and reduces variation in the throughput by up to 40%. The performance difference between Tumbler and Linux/Solaris diminishes once all the CPUs are nearly 100% utilized, i.e. the number of client threads is very high. We also evaluate Tumbler by running multiple memcached servers (see Section 5.4 for those results).

Table VII: Multiple applications. The performance improvement (%) with Tumbler over Solaris.

Mixed

FA	CA
16.4%	12.0%

BT	SW
13.8%	2.6%

FS	SC
11.3%	15.8%

CPU intensive

FR	FQ
6.0%	18.6%

SR	VL
11.5%	14.3%

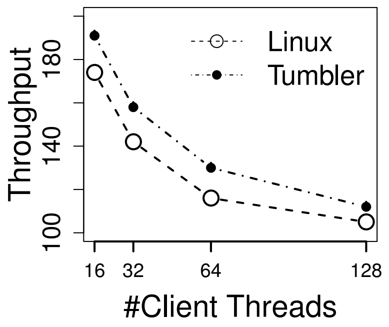
NM	ST
13.0%	19.2%

Memory intensive

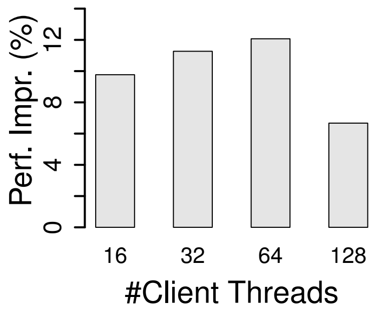
AL	FM
17.9%	14.0%

LU	OC
9.2%	8.0%

PB	JB
15.4%	11.2%

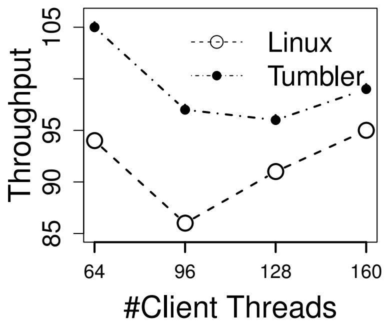


(a) Throughput.

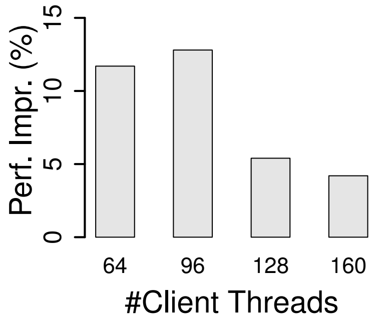


(b) Performance improvement (%).

Fig. 10: **Linux vs Tumbler**: Coscheduling multiple memcached servers. Tumbler improves throughput compared to default Linux – two memcached servers configured with **4 threads** each.



(a) Throughput.



(b) Performance improvement (%).

Fig. 11: **Linux vs Tumbler**: Coscheduling multiple memcached servers. Tumbler improves throughput compared to default Linux – two memcached servers configured with **32 threads** each.

5.4. Experiments with Multiple Applications

While the main focus of Tumbler is on improving the performance of a single multithreaded application running on a parallel machine, we also performed experiments in which we ran pairs of parallel applications simultaneously to study how Tumbler would perform in this scenario. In

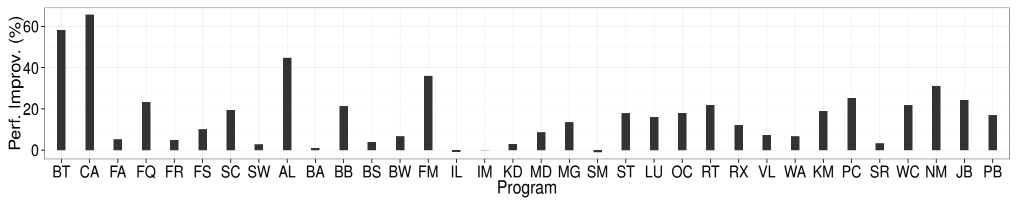


Fig. 12: Performance improvement (%) with 200% load (128 threads on 64 cores).

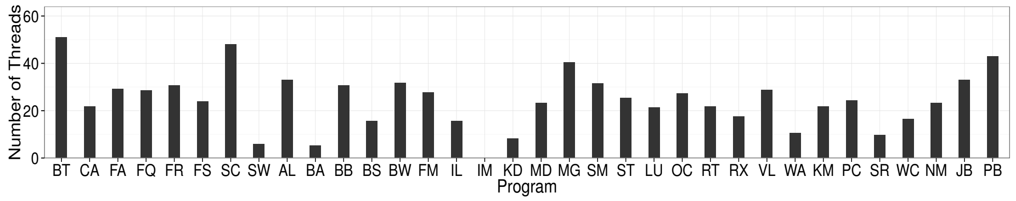


Fig. 13: The average number of threads migrated across CPUs (per second).

particular, the goal of this experiment is to see if Tumbler would simultaneously improve the performance of both applications.

On Solaris we ran combinations of compute and memory intensive application-- (FA, CA), (BT, SW), (FS, SC); pairs of compute intensive applications -- (FR, FQ), (SR, VL), (SW, FA); and pairs of memory intensive applications -- (AL, FM), (LU, OC), (PB, JB). We ran each program with 64 threads and thus for two programs a total of 128 threads were run on all 64 cores. Table VII shows that in all three cases, both applications achieve higher performance with Tumbler than Solaris. Thus, Tumbler is also effective in coscheduling multiple multithreaded programs.

On Linux, we evaluated Tumbler by running two memcached servers configured with 4 threads each and stressed the servers by varying the client threads. As Figure 10 shows, Tumbler achieves a maximum of 12% improvement in throughput compared to the default Linux.

5.5. Tumbler with Overloaded Systems

We evaluated Tumbler by running programs with 200% load (i.e., 128 threads on 64 cores). As Figure 12 shows, Tumbler achieves up to 66% performance improvement and an average of 17% improvement over Solaris. We also evaluate Tumbler by running two memcached servers with 32 threads each and varying number of client threads (64, 96, 128, 160). As Figure 11 shows, Tumbler outperforms Linux. For balancing load, Linux spreads the memcached client/server threads across the four CPUs. However, when the load is not balanced well across the 64 memcached server threads (or server threads are not fully loaded), the variation of CPU utilization across the four CPUs will be high. That is why when the number of client threads is 96, Tumbler significantly improves performance compared to Linux. In conclusion, Tumbler also gives significant performance improvement for overloaded systems.

5.6. Overhead of Tumbler

Next we analyze the overhead of Tumbler for collecting CPU utilization of individual threads and performing thread grouping. It takes 410 microseconds on average for monitoring CPU utilization values of 64 threads through the proc file system, and as we described in Section 3, it takes around 28 microseconds for identifying the groups among the 64 threads using Algorithm 2.

The cost of migrating threads during grouping phases of Tumbler is also negligible. Figure 13 shows the average number of threads migrated in a single grouping operation. This number ranges from a minimum of zero threads for IM to a maximum of 51 threads for BT. Since the variation in CPU utilization of threads of IM is below 0.05, Tumbler does not apply thread grouping. Across all the programs, on average 24.1 threads were migrated during each thread grouping operation. Since the total number of threads is 64, this represents around 38% of all 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. For every *grouping interval*, Tumbler spends around 700 microseconds (29×24.1) on changing the binding of migrated threads. Therefore, overhead of Tumbler is 1138 microseconds (1.1 ms) for 64 threads.

6. RELATED WORK

Techniques that employ thread migration to improve performance include: *load controlling techniques* [Hofmeyr et al. 2011; Johnson et al. 2010], *cache contention* [Zhuravlev et al. 2010], shared memory region [Tam et al. 2007], and *lock contention* [Lozi et al. 2012; Sridharan et al. 2006] aware thread migration.

Load controlling techniques. Hofmeyr et al. [Hofmeyr et al. 2011] propose Juggle, a pro-active load balancing technique aimed at an oversubscribed system (i.e., #threads > #cores) where #cores is not a factor of #threads on a 8-core machine (e.g., 8 threads, 16 threads, etc, on 7 cores). Instead, we focus on load imbalance issues due to input load distribution and lock contention on a 64-core machine in #threads == #cores configuration. Juggle focuses on balancing load on 1 CPU of a 8-core machine by restricting thread migrations to within the same CPU, while Tumbler balances load by migrating threads across the 4 CPUs of a 64-core machine. As authors mention, since Juggle requires barriers, it does not scale well to multiCPU systems with a large number of cores. Its implementation uses one balancer thread per core; thus, we expect it to exhibit high overhead on systems with a large number of cores.

Johnson et al. [Johnson et al. 2010] decouple load management from lock-contention management. They use blocking to control the number of runnable threads and then spinning to manage contention. [Johnson et al. 2010] mainly focuses on locking -- by automatically switching between blocking locks and spinlocks depending on contention rate. We focus on migrating threads across sockets to reduce thread latencies.

Cache contention aware thread migrations. Guided by the last-level cache miss-rates, several techniques [Blagodurov et al. 2011; Zhuravlev et al. 2010; Knauerhase et al. 2008; Merkel et al. 2010; Suh et al. 2001] use thread migrations across sockets to reduce overall LLC miss rates and improve performance. While these techniques are effective for workloads of multiple single threaded programs, they are not effective for multithreaded programs. This is because they do not consider lock contention among the threads. Tumbler improves performance when running one or more high lock contention multithreaded programs.

Shared memory region aware migration of threads. In [Thekkath and Eggers 1994] authors examine thread placement algorithms that place threads sharing memory regions on the same socket to maximize reuse. It is assumed that the shared-region information is known a priori. In [Tam et al. 2007], Tam et al. propose a thread clustering technique to detect shared memory regions dynamically. Clustering techniques group similar threads to reduce variation within clusters that leads to maximizing differences between clusters. Tumbler's grouping minimizes the difference between means of the groups and therefore its actions are exactly opposite of clustering.

Lock contention aware migration of threads. In [Lozi et al. 2012; Sridharan et al. 2006] authors observe that on multiCPU multicore systems it may be beneficial to employ thread migration to reduce the cost of acquiring locks. However, for majority of SPLASH programs the technique [Sridharan et al. 2006] yielded large performance degradation. Tumbler is superior to these solutions [Lozi et al. 2012; Sridharan et al. 2006], as these solutions require modification of either the application [Lozi et al. 2012] or the OS [Sridharan et al. 2006]. The solution we develop can be applied to any application on-the-fly.

In [Lozi et al. 2012] 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 [Lozi et al. 2012]. However, Tumbler 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.

Similarly, in [Sridharan et al. 2006] authors make the observation that in multicore multisocket 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 [Sridharan et al. 2006], for majority of SPLASH2 programs the technique frequently yielded large performance degradation. Our Tumbler technique is superior to [Sridharan et al. 2006] as it was shown to consistently provide performance improvements across a large set of multithreaded programs including SPLASH2 programs.

[Xian et al. 2008] reduces lock contention overhead by scheduling a cluster of contending threads on the same CPU. The number of threads in a cluster can be large, and in fact all threads in an application will be in the same cluster if they are synchronizing at a barrier. Thus, load is no longer balanced and parallelism is sacrificed. Tumbler maintains load balance without sacrificing parallelism. Shuffling [Pusukuri et al. 2014] minimizes lock contention by migrating threads but it does not address the load imbalance problem.

Moreover, Tumbler is superior to these solutions [Lozi et al. 2012; Sridharan et al. 2006], as these solutions require modification of either the application [Lozi et al. 2012] or the OS [Sridharan et al. 2006]. The solution we develop can be applied to any application on-the-fly.

Coscheduling Techniques. Callisto [Harris et al. 2014] is a resource management layer for parallel runtime systems. It eliminates scheduler-related interference between concurrent jobs and improves performance in coscheduled runs. ADAPT [Pusukuri et al. 2013] coschedules multithreaded applications by using machine learning techniques to characterize the interference between multithreaded applications. However, none of these works consider the load imbalance problem in multiCPU multicore systems addressed in this work.

NUMA Systems. [Brecht 1993; VMware 2005; Gupta et al. 1991; Verghese et al. 1996] studied the impact of NUMA on performance and developed adaptive scheduling techniques. Gupta et al. [Gupta et al. 1991] explore the impact of the scheduling strategies on caching behavior. Chandra et al. [Chandra et al. 1994] evaluate different scheduling and page migration policies on a multiCPU system. [Dice et al. 2015] propose locking mechanisms for improving performance on NUMA systems. [Pusukuri et al. 2011b] studied the performance impact of non-uniform load distribution across worker threads. Corey [Boyd-Wickizer et al. 2008] focuses on allowing applications to guide how shared memory data is shared between cores of a multicore system. [Pusukuri et al. 2011a] proposes a scheduling policy to reduce lock holder preemptions. Preventing lock-holder preemptions reduces the duration for which a thread *holds* the lock while our Tumbler reduces thread latency and consequently reduces lock acquisition latencies.

7. CONCLUSIONS

We introduced Tumbler, a load balancing technique that continuously monitors CPU utilization of threads of a multithreaded program and adaptively migrates threads across CPUs to balance load and thus reduce thread latencies. Tumbler improves performance on both Solaris 11 (maximum 37% and on average 12%) and Linux (maximum 26% and on average 10%). Our portable implementation of Tumbler does not require changes to the application or the OS kernel.

REFERENCES

BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. PACT '08. ACM, New York, NY, USA, 72–81.

BLAGODUROV, S., ZHURAVLEV, S., DASHTI, M., AND FEDOROVA, A. 2011. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. USENIXATC'11. USENIX Association, Berkeley, CA, USA, 1–1.

BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. 2008. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. USENIX Association, Berkeley, CA, USA, 43–57.

BRECHT, T. 1993. On the importance of parallel application placement in numa multiprocessors. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*. Sedms'93. USENIX Association, Berkeley, CA, USA, 1–1.

CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. 2004. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*. ATEC '04. USENIX Association, Berkeley, CA, USA, 2–2.

CHANDRA, R., DEVINE, S., VERGHESE, B., GUPTA, A., AND ROSENBLUM, M. 1994. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*. ASPLOS-VI. ACM, New York, NY, USA, 12–24.

CURTSINGER, C. AND BERGER, E. D. 2013. Stabilizer: Statistically sound performance evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. ACM, New York, NY, USA, 219–228.

DATACACHE. 2012. Data cache benchmark. http://parsa.epfl.ch/cloudsuite/memcached.html.

DICE, D., MARATHE, V. J., AND SHAVIT, N. 2015. Lock cohorting: A general technique for designing numa locks. *ACM Trans. Parallel Comput. 1,* 2, 13:1–13:42.

FERDMAN, M., ADILEH, A., KOCBERBER, O., VOLOS, S., ALISAFAEE, M., JEVDJIC, D., KAYNAK, C., POPESCU, A. D., AILAMAKI, A., AND FALSAFI, B. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. ACM, New York, NY, USA, 37–48.

FITZPATRICK, B. 2004. Distributed caching with memcached. *Linux J. 2004,* 124, 5.

GUPTA, A., TUCKER, A., AND URUSHIBARA, S. 1991. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. *SIGMETRICS Perform. Eval. Rev. 19*, 120–132.

HARRIS, T., MAAS, M., AND MARATHE, V. J. 2014. Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. ACM, New York, NY, USA, 24:1–24:14.

HOFMEYR, S., COLMENARES, J. A., IANCU, C., AND KUBIATOWICZ, J. 2011. Juggle: Proactive load balancing on multicore computers. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*. HPDC '11. ACM, New York, NY, USA, 3–14.

JOAO, J. A., SULEMAN, M. A., MUTLU, O., AND PATT, Y. N. 2012. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. ACM, New York, NY, USA, 223–234.

JOHNSON, F. R., STOICA, R., AILAMAKI, A., AND MOWRY, T. C. 2010. Decoupling contention management from scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. ASPLOS '10. ACM, New York, NY, USA, 117–128.

KNAUERHASE, R., BRETT, P., HOHLT, B., LI, T., AND HAHN, S. 2008. Using os observations to improve performance in multicore systems. *IEEE Micro 28,* 3, 54–66.

LOZI, J.-P., DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. 2012. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. USENIX Association, Berkeley, CA, USA, 6–6.

MCDOUGALL, R. AND MAURO, J. 2006. *Solaris Internals, second edition*. Prentice Hall, USA.

MCDOUGALL, R., MAURO, J., AND GREGG, B. 2006. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Prentice Hall, USA.

MENDELSON, A. AND GABBAY, F. 2001. The effect of seance communication on multiprocessing systems. *ACM Trans. Comput. Syst. 19,* 2, 252–281.

MERKEL, A., STOESS, J., AND BELLOSA, F. 2010. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*. EuroSys '10. ACM, New York, NY, USA, 153–166.

PBZIP2. 2001. Pbzip2. http://compression.ca/pbzip2/.

PETER, S., SCHÜPBACH, A., BARHAM, P., BAUMANN, A., ISAACS, R., HARRIS, T., AND ROSCOE, T. 2010. Design principles for end-to-end multicore schedulers. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*. HotPar'10. USENIX Association, Berkeley, CA, USA, 10–10.

PUSUKURI, K. K., GUPTA, R., AND BHUYAN, L. N. 2011a. No more backstabbing... a faithful scheduling policy for multithreaded programs. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. PACT '11. IEEE Computer Society, Washington, DC, USA, 12–21.

PUSUKURI, K. K., GUPTA, R., AND BHUYAN, L. N. 2011b. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*. IISWC '11. IEEE Computer Society, Washington, DC, USA, 116–125.

PUSUKURI, K. K., GUPTA, R., AND BHUYAN, L. N. 2013. Adapt: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim. 9,* 4, 45:1–45:24.

PUSUKURI, K. K., GUPTA, R., AND BHUYAN, L. N. 2014. Shuffling: A framework for lock contention aware thread scheduling for multicore multiprocessor systems. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT '14. ACM, New York, NY, USA, 289–300.

SPECOMP. 2001. http://www.spec.org/omp.

SRIDHARAN, S., B. KECK, R. M., AND S. CHANDRA, P. K. 2006. Thread migration to improve synchronization performance. In Workshop on Operating System Interference in High Performance Applications.

SUH, G. E., RUDOLPH, L., AND DEVADAS, S. 2001. Effects of memory performance on parallel job scheduling. In *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*. JSSPP '01. Springer-Verlag, London, UK, UK, 116–132.

TAM, D., AZIMI, R., AND STUMM, M. 2007. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. ACM, New York, NY, USA, 47–58.

THEKKATH, R. AND EGGERS, S. J. 1994. Impact of sharing-based thread placement on multithreaded architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*. ISCA '94. IEEE Computer Society Press, Los Alamitos, CA, USA, 176–186.

VERGHESE, B., DEVINE, S., GUPTA, A., AND ROSENBLUM, M. 1996. Operating system support for improving data locality on cc-numa compute servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VII. ACM, New York, NY, USA, 279–289.

VMWARE. 2005. Vmware esx server 2 numa support. white paper. http://www.vmware.com/pdf/esx2_NUMA.pdf.

WENTZLAFF, D. AND AGARWAL, A. 2009. Factored operating systems (fos): The case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev. 43,* 2, 76–85.

WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. 1995. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*. ISCA '95. ACM, New York, NY, USA, 24–36.

XIAN, F., SRISA-AN, W., AND JIANG, H. 2008. Contention-aware scheduler: Unlocking execution parallelism in multithreaded java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. OOPSLA '08. ACM, New York, NY, USA, 163–180.

YOO, R. M., ROMANO, A., AND KOZYRAKIS, C. 2009. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*. IISWC '09. IEEE Computer Society, Washington, DC, USA, 198–207.

ZHANG, E. Z., JIANG, Y., AND SHEN, X. 2010. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '10. ACM, New York, NY, USA, 203–212.

ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. ASPLOS '10. ACM, New York, NY, USA, 129–142.