Contention-Aware Scheduler: Unlocking Execution Parallelism in Multithreaded Java Programs

Conference Paper *in* ACM SIGPLAN Notices · September 2008

DOI: 10.1145/1449764.1449778 · Source: DBLP

Witawas Srisa-an
University of Nebraska at Lincoln
101 PUBLICATIONS **1,248** CITATIONS
SEE PROFILE

Hong Jiang
University of Texas at Arlington
344 PUBLICATIONS **6,292** CITATIONS
SEE PROFILE

Contention-Aware Scheduler: Unlocking Execution Parallelism in Multithreaded Java Programs

Feng Xian, Witawas Srisa-an, and Hong Jiang

Department of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
{fxian,witty,jiang}@cse.unl.edu

Abstract

In multithreaded programming, locks are frequently used as a mechanism for synchronization. Because today's operating systems do not consider lock usage as a scheduling criterion, scheduling decisions can be unfavorable to multithreaded applications, leading to performance issues such as convoying and heavy lock contention in systems with multiple processors. Previous efforts to address these issues (e.g., transactional memory, lock-free data structure) often treat scheduling decisions as "a fact of life," and therefore these solutions try to cope with the consequences of undesirable scheduling instead of dealing with the problem directly.

In this paper, we introduce *Contention-Aware Scheduler* (*CA-Scheduler*), which is designed to support efficient execution of large multithreaded Java applications in multiprocessor systems. Our proposed scheduler employs a scheduling policy that reduces lock contention. As will be shown in this paper, our prototype implementation of the CA-Scheduler in Linux and Sun HotSpot virtual machine only incurs 3.5% runtime overhead, while the overall performance differences, when compared with a system with no contention awareness, range from a degradation of 3% in a small multithreaded benchmark to an improvement of 15% in a large Java application server benchmark.

Categories and Subject Descriptors D.3.4 [*Programming Language*]: Processors—Run-time Environments; D.4.1 [*Operating System*]: Process Management—Concurrency, Scheduling, Synchronization, Threads

General Terms Experimentation, Languages, Performance

1. Introduction

To support portability, better security, and ease of resource management, modern object-oriented languages such as Java and C# utilize virtual machine technologies to provide execution environments by emulating simple instruction sets such as Java bytecodes or the .NET intermediate language. Because these virtual machines often provide complete execution environments, they also generate rich runtime information during execution. Past studies have shown that a virtual machine often exploits such information to further optimize itself in subsequent runs or continuously during execution, especially in long-running programs.

For example, method invocation information has been used by VMs to select the complexity of dynamic compilation optimizations [18]. Runtime behaviors have been used to select the optimal garbage collection technique for an application or an execution phase within an application [28]. Moreover, logging of runtime information has been valuable in helping programmers detect and isolate errors as well as identify performance bottlenecks.

Yet, such information has rarely been exploited by the underlying operating systems to create more efficient environments for high-level language executions. To date, the usage of this information has been limited to garbage collection tuning through virtual memory managers [12, 37]. We see a great opportunity to leverage this information to further improve the efficiency of thread-level parallelism, a programming paradigm widely adopted today due to the emerging popularity of chip multiprocessor systems.

In multithreaded applications, locks are frequently used as a mechanism to prevent multiple threads from having concurrent access to shared resources and to synchronize the execution order of threads. For example, a lock can be used to protect a shared code section, commonly referred to as a *critical section*. When a thread wants to access this code section, it must first obtain the lock. If it is successful, it can perform memory access. If it is not successful (i.e., lock contention[1]), it either keeps trying to obtain the lock (spin-locking) or is suspended until the lock becomes available. Because today's operating systems do not consider lock usage as a scheduling criterion, they can:

- *schedule threads that contend for currently locked resources.* In this scenario, multiple threads that try to access the same critical section can be scheduled to run at the same time on different processors. The one that successfully obtains the lock continues to execute, while the ones that fail to obtain the lock are suspended.
- *preempt an executing thread in the middle of a critical section.* In this scenario, a thread holding the lock to the critical section is suspended due to time quantum expiration. This means that if the operating system schedules other threads that need to access the same critical section, they will also be suspended (i.e., convoyed).

While there is a large body of work that addresses the issue of lock contention by eliminating or minimizing the use of locks (e.g., transactional memory and lock-free data structures [16, 24]), the focus of our work is entirely different. Specifically, our research goal is *not to avoid using locks, but instead to proactively avoid lock contention by supplying the necessary runtime information to the underlying operating system so that it can make better scheduling decisions*. Our rationale for taking this approach is outlined below:

1. It is conceivable that more informed scheduling decisions can reduce the number of lock contention, leading to greater execution parallelism and improved overall performance. However, it is unclear whether such improvement outweighs the additional scheduling complexity that may result in longer and non-deterministic decision times. Over the past few years, we have seen that in certain types of applications, it is worthwhile to trade some bottom-line performance for higher programming productivity (e.g., the adoption of garbage collection and transactional memory), better security and correctness, and greater portability (e.g., the adoption of virtual machine monitors and high-level language virtual machines). Based on a similar argument, our work investigates whether it is worthwhile to trade the simplicity of modern schedulers for higher execution parallelism.
2. It is common for hardware components supporting large-enterprise Java applications to be very specialized (e.g., a large number of processing cores, large storage space, and wide network bandwidth). However, these systems frequently utilize generic commercial operating systems. It is debatable whether the stringent performance requirements (e.g., high throughput and short response time) of these server applications warrant customized operating systems and virtual machines. As will be shown in this paper, our implementation effort to extend a widely-adopted operating system (OS) and a commercial Java Virtual Machine (JVM) to achieve our research goal is quite modest, while yielding significant performance improvements. As a result, the integration of our proposed solution into commercial operating systems is worth contemplating.
3. There are many deployed Java and .NET based programs that already use locks to manage concurrency. Our proposed solution will allow these applications to immediately take advantage of these benefits without having to rewrite the application code.

[1] A *lock contention* occurs whenever one process or thread attempts to acquire a lock held by another process or thread [26].

This paper. We introduce a *Contention-Aware Scheduler* or CA-Scheduler. Our proposed scheduler receives information directly from the JVM to proactively reduce the *possibility of lock contention* by

1. dynamically clustering threads that share similar lock-protected resources, and then serializing each cluster to reduce the likelihood of lock contention, and
2. giving longer execution quanta and higher execution priority to threads in the middle of critical sections.

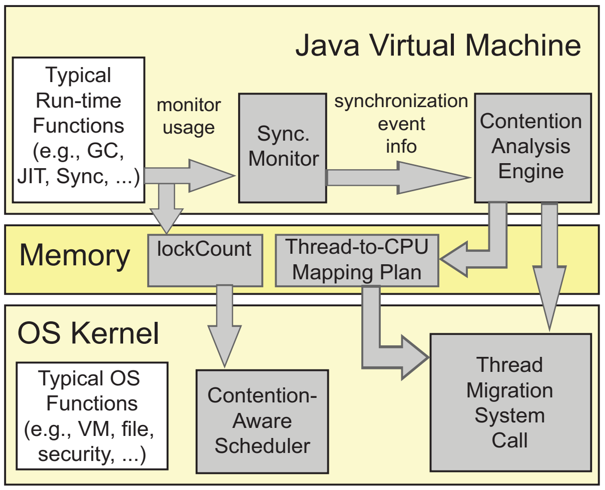
We implemented a prototype of *CA-Scheduler* as part of a multiprocessor version of the Linux scheduler with necessary support from Sun HotSpot JVM (presented in Section 2). Our analysis of the complexity of the CA-Scheduler showed that it incurred only 2 to 3.5% runtime overhead (presented in Section 3). We then evaluated its performance using seven multithreaded Java benchmarks in a computer system with 16 processors. Our results showed that the differences in the overall performance ranged from a degradation of 3% in a desktop application with low lock contention to an improvement of 15% in a large application server benchmark with high lock contention. The reductions in the number of lock contention are 38% and 45% in two Java applications server benchmarks, ECPerf and *jAppServer2004*, respectively (presented in Section 4).

2. Introducing CA-Scheduler

In this section, we briefly describe an overview of the proposed CA-Scheduler, discuss its design goals, and provide the implementation details.

2.1 Overview

Recently, we have seen significant advancements of virtual machine technologies, ranging from high-level language virtual machines such as JVM and .NET CLR to system virtual machines such as Microsoft Virtual PC [20] and Transmeta Code Morphing [10]. One common characteristic of these virtual machines is that they are very dynamic. Various profiling and optimization techniques are used to improve performance. For example, most JVMs often monitor garbage collection performance and allocation rates to decide if the heap size should be adjusted. Moreover, modern JVMs monitor lock-contention behavior to adaptively apply different locking mechanisms to maintain good performance and improve CPU utilization [4, 11]. Unfortunately, such runtime information is usually discarded at the end of execution or is used only within virtual machines [3, 27, 28].



Java Virtual Machine
Typical Run-time Functions (e.g., GC, JIT, Sync, ...)
monitor usage
Sync. Monitor
synchronization event info
Contention Analysis Engine
Memory
lockCount
Thread-to-CPU Mapping Plan
OS Kernel
Typical OS Functions (e.g., VM, file, security, ...)
Contention-Aware Scheduler
Thread Migration System Call

Figure 1. An Overview of the proposed CA-Scheduler

We believe that this runtime information is also valuable to operating systems. However, there have been very few efforts that allow operating systems to exploit such information. Two notable examples are efforts by Yang *et al.* [37] and Grzegorczyk *et al.* [12] to exploit virtual memory information to adaptively identify optimal heap sizes to improve garbage collection performance. Another example is an effort by Xian et al. to exploit object allocation behavior as a scheduling criterion to reduce garbage collection overhead [36]. Based on the results of these efforts, we see a great opportunity to make operating systems exploit runtime information generated by virtual machines in order to control thread execution orders to avoid lock contention.

Many modern schedulers make decisions according to the amount of time a thread is spent on the CPU and the amount of time a thread is spent on doing I/O accesses [1]. To support our proposed technique, we created three additional components, two in the HotSpot JVM and one in the Linux kernel, to assist with making contention-aware scheduling decisions (see Figure 1): a *Synchronization Monitor*, which is implemented in the JVM to record synchronization activities; a *Contention Analysis Engine*, which is also implemented in the JVM to analyze the lock usage information and create CPU mapping plans that will be used by our modified kernel; and a *Thread Migration* system call, which is implemented in the kernel to migrate threads to the corresponding CPUs based on the JVM generated plan.

We also modified a typical scheduler to consider lock usage as a scheduling criterion (i.e., the Contention-Aware Scheduler in Figure 1). The modified JVM passes information to the kernel through two variables that contain the *Thread-to-CPU Mapping Plan* and the number of monitors held by the currently executing thread (i.e., *lockCount* in Figure 1). The next two subsections provide detailed information about the design and implementation of each of these components.

2.2 Design Goals

The proposed CA-Scheduler is designed to reduce the occurrences of lock contention in large multithreaded Java applications by allowing our modified Java Virtual Machine to "push" pertinent runtime information to our modified operating system so that it can accomplish two scheduling tasks:

Task 1—*Preventing contention due to concurrent execution of multiple threads on multiple processors.* When threads that share a lock-protected resource are executing in parallel, they may contend on that resource, causing threads that cannot obtain the lock to be suspended. (We refer to this type of contention as *inter-processor contention*.) One possible solution is to *not schedule* threads that may contend. To do so, the scheduler must know exactly when a thread is holding the lock and precisely which threads share that same lock.

In typical multiprocessor schedulers such as the one employed in Linux, a thread can be scheduled on different processors throughout its lifetime to achieve load balancing. Thus, dynamically maintaining the necessary information to achieve our goal is likely to incur significant runtime overhead. A study has shown that propagating such information to every processor can be quite expensive. Moreover, propagation delays can also result in unstable system states. [35].

We modified Sun HotSpot JVM to cluster threads based on lock usage. In our approach, each cluster contains *a group of threads that are likely to contend with each others*. The clustering and thread-to-CPU mapping information is then made available to the operating system kernel for the actual CPU assignment (more information about this process is provided in Sections 2.3.1 and 2.3.2). By executing threads that share *lock-protected resources* (from now on, referred to as *shared resources*) serially, we eliminate the possibility that multiple threads executing on different processors access a shared resource simultaneously. Note that our study also revealed that there are instances in which threads share resources across clusters (e.g., some resources are shared by most threads or all threads). In such a scenario, contention may be unavoidable.

One possible shortcoming of this approach is underutilization of processing cores. The execution of threads may be serialized even though there are enough processing units to support greater parallelism. As will be shown in Section 4.2, we found that in a server application utilizing the same number of threads as the number of processors, it is possible that *the performance benefit of reducing contention is greater than the performance benefit of higher parallelism*.

In addition, in large application server applications, the number of threads tends to be significantly larger than the number of processing cores; thus, proactively serializing threads still allows processors to be well utilized. Lastly, our system also has a load-balancing mechanism to equally divide some special clusters to multiple CPUs (more information about the load balancing mechanism is provided in Section 2.3.1).

Task 2—*Avoiding thread preemption while locks are being held.* With clustering, we can reduce lock contention due to concurrently executing threads. However, in most schedulers, a thread can be preempted while it is holding locks, leading to contention when other threads sharing the same resources are scheduled. (We refer to this type of contention as *intra-processor contention*.) To reduce this type of contention, we adopted a quantum-renewing approach that gives longer execution quanta to threads in critical sections [34, 2, 19]. Our scheduler checks if an executing thread is holding a lock. If it is, the scheduler ignores the timer interrupt and schedules the thread for one more quantum. If the renewed quantum expires before the thread relinquishes the lock, the scheduler can make more quantum renewals.

To prevent other threads from starving, our scheduler can only make at most a predefined number of consecutive renewal requests for an executing thread. We discovered that in Java, a thread already holding a lock sometimes acquires several more locks. This means that the time taken to release the first lock can be very long. Moreover, suspended threads that hold multiple locks can cause other threads to convoy. Thus, it is more beneficial to schedule these threads first so that locks are relinquished quickly. We propose a dynamic prioritization strategy called *Critical-Section-First* (*CSF*) to schedule suspended threads that are holding multiple locks at higher priority. With this approach, we can significantly reduce occurrences of convoying. Moreover, our approach also shortens the periods that threads must stay in the suspended state waiting for locks to be released. In the next subsections, we detail the necessary modifications made to the JVM and the OS to support the proposed tasks. We also provide information related to tuning the proposed system and analyze its overhead.

2.3 Implementation Details

We implemented the CA-Scheduler in the HotSpot JVM, which is released as part of Sun's OpenJDK project (version 7)[2] and the Linux multiprocessor kernel version 2.6.20. In the rest of this section, we describe modifications made to HotSpot (from now on, referred to as *JVM*) and the Linux kernel (from now on, referred to as *kernel*) to support the proposed CA-Scheduler.

2.3.1 Java Virtual Machine Modifications

To provide the necessary information to the kernel, we modified the JVM to perform the following functions:

Record synchronization events. We modified HotSpot to periodically record synchronization events for both interpretation and dynamic compilation modes. We captured these events by augmenting the functions inside HotSpot that handle monitors and manipulate locks. Our recording mechanism is activated once the number of created threads is greater than a predefined value. In our experiment, we set this number to 10 because our smallest benchmark spawns 10 threads. Once the number of created threads has not changed for a long period of time, the mechanism is deactivated. The mechanism can be reactivated if the number of threads changes again.

When a thread enters a monitor, the JVM inserts the object associated with the monitor into the *synchronization-event vectors* (*seVectors*). Each thread has an associated vector. To provide flexibility in choosing the amount of storage overhead, the size (referred to as *s*) of *seVector* can be tuned. We will discuss the empirical process to identify a generalized value of s in Section 3.2.

Form clusters of related threads. There are two steps in the clustering process. Once the number of synchronization events of a thread has reached s (i.e., one of the $seVectors$ has the size of *s*), the JVM performs the first step by processing each *seVector* to generate a corresponding *contention vector* (*conVector*). Each entry of a *conVector* indicates the number of times that a thread has attempted to access a shared object (i.e., the number of synchronization events performed on the object). An example of converting *seVectors* to *conVectors* is provided in Figure 2.

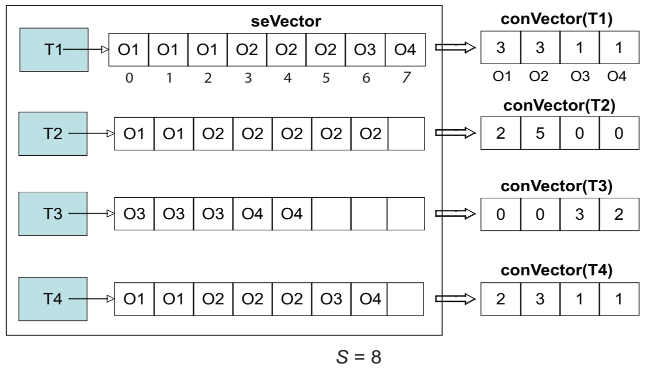


Figure 2. Converting *seVectors* to *conVectors*. In this example, we assume that $s = 8$. When a $seVector$ is fully utilized (that of T1 in this example), all $seVectors$ are processed to generate $conVectors$.

[2] See OpenJDK project at http://openjdk.java.net.

The second step is *clustering threads*. There are many existing clustering algorithms such as K-means Fuzzy C-means and hierarchical clustering [5, 15, 6]. However, these algorithms are too expensive to be deployed on-line in large multithreaded systems. Another option is to employ offline profiling, which applies clustering algorithms to trace information generated during the previous runs [9]. This clustering information is then used to perform various optimizations in the subsequent runs. While profiling can be accurate, assuming that the runtime behavior does not change from one run to another, it requires an overhead of executing the profile run. If the runtime behavior changes, the profiled information may no longer be useful.

To perform clustering accurately and cheaply, we propose an on-line heuristic clustering algorithm. The algorithm is based on an assumption that clusters can be formed in such a way that threads belonging to two different clusters seldom contend, except for some objects that are shared by a large number of threads (refer to as globally-shared objects). This assumption is based on the result of our previous work on thread clustering [9]. Our algorithm considers an object to be globally shared if more than half of the total number of threads has tried to acquire the monitor associated with this object. When forming clusters, these globally-shared objects are not considered as clustering criteria. Typically, the number of globally-shared objects is very small (see Table 1).

Our clustering mechanism works as follows. First, the JVM randomly chooses a thread as a representative of a cluster. It then calculates the *similarity* of each of the remaining threads to this thread. The *similarity* of two threads, T_1 and T_2, is a *normalized vector product* that can be calculated using the following formula:

$$similarity(T_1, T_2) = \frac{1}{s^2} \sum_{i=1}^{v} T_1[i] \times T_2[i],$$

where $T_x[i]$ is the i^{th} entry of the *conVector* of thread T_x and v is the size of the *conVector*.

As defined earlier, s is the size of the *seVector*. Thus, the sum of a *conVector* is equal to s, resulting in *similarity* values ranging from 0 to 1. Also note that the JVM disables the recording mechanism when it tries to form clusters. This is necessary to maintain consistency in all *seVector*s.

The rationale for choosing the proposed similarity metric is two-fold. First, it only considers the same object entries in both vectors with non-zero values. A non-zero value means that these two threads (T_1 and T_2) access the same objects and therefore can *possibly contend on the same object*. Second, the degree of similarity between T_1 and T_2 also indicates the likelihood that T_1 and T_2 will contend on an object. That is, a higher *similarity* value means that the likelihood of contention is also higher.

If the *similarity* between the representative thread and a thread is greater than *simThreshold*, then this thread is placed in the same cluster as the representative thread. (The tuning of *simThreshold* will be discussed in Section 3.2.) The process is repeated until every thread has been compared with the representative thread. At this point, the first cluster is formed. Our JVM then repeats the same process on the remaining threads that are not part of any cluster, until no more clusters can be formed.

The computation complexity of the clustering algorithm is $O(N_t \times N_c)$, where N_t is the total number of Java threads and N_c is the total number of clusters. Generally, N_c is much smaller than N_t. Also note that it is possible that selecting a different representative thread can yield a different clustering result. As will be shown in Section 3.2, the proposed clustering scheme provides a good balance between accuracy (over 75% in all applications when compared to manually-formed clusters) and low clustering overhead (less than 2% of overall execution time in all applications). This high degree of accuracy indicates that the differences due to randomness in selecting the representative thread may not be significant.

While forming clusters, our JVM also calculates the *contention intensity* of each cluster by averaging all the *similarity* values within that cluster. This value is then used to classify whether a cluster is strongly contended (referred to as *strong-contention*) or weakly contended (referred to as *weak-contention*). This classification is used to help with the load balancing process, in which our JVM generates information to map clusters to CPUs.

Generate CPU mapping plan. Once clusters are formed, our JVM creates a mapping plan that achieves load balancing and minimizes contention across multiple processors. There are three steps in this process. In the first step, our JVM segregates clusters into two groups: 1) a *strong-contention group* that comprises strong-contention clusters, and 2) a weak-contention group that comprises weak-contention clusters. Specifically, if the *contention intensity* of a cluster is higher than the *ciThreshold*, the cluster is placed in the strong-contention group. If the intensity is less than *ciThreshold*, the cluster is placed in the weak-contention group (more information about tuning *ciThreshold* is given in Section 3.2).

Next, the JVM calculates the number of threads that should be assigned to each processor in an ideal case. As stated earlier, N_c represents the number of clusters and N_t represents the number of threads. The number of processors is denoted as N_p. Ideally, there should be $T_{avg} = \frac{N_t}{N_p}$ threads assigned to each processor. Suppose that processors are represented as $P_1, P_2, \ldots, P_{N_p}$ and their workloads (i.e., the number of threads) are $TP_1, TP_2, \ldots, TP_{N_p}$. We denote a sorted list of underutilized processors as U. A processor is underutilized if its workload is less than T_{avg}. Since the initial workload of each processor with respect to our application is 0, U is initially equal to $P_1, P_2, \ldots, P_{N_p}$. Note that U will dynamically change throughout this process.

In the second step, the JVM maps clusters in the strong-contention group to the processors. Suppose that there are m strong-contention clusters. Our JVM assigns these clusters to processors in the following manner. First, these m clusters are sorted from the largest size to the smallest size, which are denoted as $SC_1, SC_2, \ldots, SC_m$, respectively. The numbers of threads in these clusters are $TSC_1, TSC_2, \ldots, TSC_m$. Then, the JVM assigns the largest cluster SC_1 to P_1 and updates the workload of P_1 (i.e., $TP_1 \Leftarrow TSC_1$). At the same time, it also updates U by removing any processor that has a workload equal to or greater than T_{avg}. Once these processors are removed, U is rearranged in the order of the lightest workload to the heaviest workload.

In the third step, the JVM performs load balancing with weak-contention clusters. Assume that the current size of U is n; i.e., there are n processors that are underutilized. We denote these processors as $PU_1, PU_2, \ldots, PU_n$ in an increasing order of workload, which are $TU_1, TU_2, \ldots, TU_n$, respectively. Note that at this point, the remaining processors $(N_p - n)$ should be fully utilized with strong-contention clusters and are not considered as available processors in the current step.

Suppose that there are k clusters in the weak-contention group. They are denoted as $WC_1, WC_2, \ldots, WC_k$ from the largest size to the smallest size. The numbers of threads in these clusters are $TWC_1, TWC_2, \ldots, TWC_k$, respectively. To achieve load balancing, there should be $T_{avg2} = \frac{\sum_{i=1}^{n} TU_i + \sum_{i=1}^{k} TWC_j}{n}$ threads assigned to each of the remaining processors. The load balancing process simply merges or splits each of the weak-contention clusters until the workload of each underutilized processor is equal to T_{avg2}. If some threads become inactive over time, it may become necessary to regenerate a new mapping plan to rebalance the load.

In terms of time complexity, the first step (segregating clusters into strong- and weak-contention groups) has the complexity of $O(N_c)$. The second step (strong-contention mapping) has two major tasks: sorting cluster ($O(m \times log\ m)$) and updating U ($O(m \times log\ N_p)$). Thus, the complexity of the second step is $O(m \times log\ m + m \times log\ N_p)$. For the last step, the time complexity is $O(k \times log\ m + k \times log\ N_p)$. As a result, the total time complexity of clustering and making the thread-to-processor mapping plan is $O(N_c \times log\ N_c + N_c \times log\ N_p)$ because $(m + k) = N_c, (m \times log\ m + k \times log\ k) < (m + k) \times log\ (m + k)$ or $N_c \times log\ N_c$.

Mark when threads are in critical sections. For each Java thread, our JVM maintains a field called *lockCount*, which indicates the number of unique locks held by a thread. When a thread has successfully entered a monitor, our JVM increments its *lockCount* value. When a thread exits a monitor, our JVM correspondingly decrements its *lockCount* value. Therefore, we can use the value of *lockCount* to check whether a thread is holding locks. If the value is 0, then the thread is not holding any lock. If it is 1, the thread is holding a lock. If it is greater than 1, then the thread is holding multiple locks. This information will be used by our proposed *Critical-Section-First* (*CSF*) scheduler to dynamically adjust the execution priority of a thread. More information about the *CSF* scheduler will be given in the next subsection.

2.3.2 Kernel Modifications

In this section, we describe modifications made to the kernel to (i) segregate Java threads from other types of threads; (ii) physically assign threads to CPUs; and (iii) make the scheduler contention-aware.

Create a system call to register Java threads. The proposed CA-Scheduler is built on the top of the default Linux scheduler. Thus, these two scheduling policies can co-exist, and the CA-Scheduler can selectively apply its scheduling policy to the corresponding Java threads. However, in open systems such as Linux, there are also non-Java threads (e.g., essential services) concurrently running with Java threads. To distinguish between Java and non-Java threads, we created a new system call, *register_thread*, to allow Java threads to set a field, *isJava* in the thread data structure (i.e., *task_struct* in Linux).

When a Java thread is executing, timer interrupts still occur as in the default scheduler; however, the CA-Scheduler can choose to ignore the interrupts and continue to renew more time slices as needed. It is also likely that a system may have multiple Java applications running at the same time. These Java applications may require different configurations of the CA-Scheduler (e.g., different parameter values). The proposed CA-Scheduler is designed to be customizable so that each Java application can configure its own scheduling parameters. To achieve this flexibility, we modified HotSpot to create a metadata structure for each VM instantiation. The structure is instantiated through a new system call, *register_vm*, during initialization and is referenced by the CA-Scheduler through a pointer field, *jvm_info* in *task struct* to enforce the corresponding policy when a thread belonging to a Java application is executed.

Create a system call to map threads to CPUs. Because our mechanism to map thread to CPU is an on-line one, our JVM does not initially have the mapping information as it is incrementally generated during runtime. Therefore, at the beginning of execution, the default thread scheduler is used. Once the JVM has generated the mapping plan, threads may have to be migrated to different CPUs based on this plan. To accomplish this task, we created a system called *migrate_cluster* to retrieve the mapping plan from the JVM (discussed in Section 2.3.1) and to physically assign threads to CPUs. Note that each Java thread is pinned to the assigned processor until the next mapping call to ensure that the default load-balancing mechanism used in Linux does not interfere with our mapping. Note that our system can create multiple mapping plans throughout execution of an application.

Modify thread scheduler. After threads have been mapped to CPUs, the kernel independently schedules threads assigned to each processor. To reduce the occurrences of intra-processor contention, we propose a *Critical-Section-First* (*CSF*) scheduling strategy that makes threads relinquish held locks as soon as possible. This minimizes the waiting time of other threads to obtain these held locks. There are two major components of the proposed Critical-Section-First scheduling strategy:

Dynamic prioritization mechanism. To schedule threads in critical sections first, we need to prioritize the threads that have larger *lockCount* values. In our implementation, we extended the Linux scheduler to dynamically adjust the priority of each thread according to its *lockCount*. The default Linux scheduler uses a priority-based round-robin scheduling policy and adjusts thread priority based on the average sleeping time. We adopted the existing mechanism in Linux to perform priority adjustment. That is, each thread can get a bonus to increase its priority. The calculations of the bonus and priority adjustment are as follows:

$$bonus = (p \rightarrow lockCount) \times \frac{MAX_{bonus}}{MAX_{lockCount}}$$

$$prio = p \rightarrow static\_prio - bonus$$

This formula indicates that the bonus is proportional to the *lockCount* value. It maps a thread's *lockCount* to the range of 0 to MAX_{BONUS}. If a thread is not in any critical section (i.e., *lockCount*=0), then its priority does not change. Otherwise, its static priority is subtracted, meaning that its priority rises. Note that $bonus$ is between -5 and +5. By prioritizing threads in critical sections, we also avoid priority inversion, in which a suspended thread with low priority holds a lock that is needed by a higher priority thread [26].

Quantum renewal mechanism. We adopted a quantum renewing mechanism that allows a thread in a critical section to continue execution until it exits the critical section [2, 19, 34]. Our scheduler checks the executing thread's *lockCount* when a timer interrupt occurs. If its *lockCount* is greater than 0, our scheduler ignores the interrupts and reschedules the thread for another quantum. To avoid starvation and live-lock, we set the maximal number of renewals (*maxRenewal*) to five. That is, a thread will be preempted after the scheduler has made five consecutive renewal requests. *We selected five based on our empirical study that shows that a thread rarely needs more than five quanta to acquire and relinquish a lock.*

2.4 Summary

The proposed CA-Scheduler is designed to reduce lock contention in multithreaded Java applications through a collaboration between our modified HotSpot JVM and our modified Linux kernel. We extended HotSpot to include the following services: (i) monitoring synchronization events; (ii) processing the recorded information and performing analysis to cluster threads that are likely to contend; and (iii) generating thread-to-CPU mapping plans that achieve contention reductions and load-balancing. We then extended the Linux kernel to include the following features: (i) being able to receive thread-to-CPU mapping plans from the JVM; (ii) executing the mapping plans and enforcing the contention-aware policy; (iii) being able to fall back to the default policy when native threads are scheduled or when the contention-aware policy is not applicable; and (iv) having escape mechanisms to avoid starvation and livelock. The proposed system also contains several parameters that users can tune to yield effective contention reductions.

3. Tuning and Overhead Analysis

The multi-core system used in our experiment has eight Dual-Core AMD Opteron processors (16 processors in total). The system has 32GB of physical memory. We chose four client-side benchmarks: *eclipse*, *hsqldb*, *lusearch*, and *xalan* from the DaCapo suite [7][3]. We did not include any single-threaded benchmark because our proposed contention-aware policy is not activated when running single-threaded Java programs. We also chose three server-side benchmarks: *jbb2005* (from SPEC), *jAppServer2004* (from SPEC), and *ECPerf* (from Sun Microsystems). It is worth noting that we initially set the size of Java heap to 64MB and allowed it to freely grow up to 2GB as needed by the applications.

We executed the five stand-alone benchmarks (*eclipse*, *hsqldb*, *lusearch*, *xalan*, and *jbb2005*) on the above system with unmodified HotSpot and unmodified kernel (from now on, referred to as *default*) and the CA-Scheduler-enabled (from now on, referred to as *CASe*). As mentioned earlier, HotSpot is released as part of the OpenJDK 7 project and the Linux kernel is Version 2.6.20.

For *jAppServer2004* and *ECPerf*, we set up a real-world Java application server environment. The environment consists of three machines: application server, database server, and client. These machines are connected through an internal private network to minimize latencies due to network congestion. The application server is the above system configured to run *default* and *CASe*. The server machine is also configured with no other major services (e.g., DNS, mail, database) to create a real-world scenario where the application server is *not* likely to be used for other major services. The database server is a Sun Blade with dual 2GHz AMD

[3] The version of DaCapo benchmarks that we used is dacapo-2006-10.

Benchmark	Workload size	Number of threads	Shared objects	Number of globally contended objects	Contention per object		
					average	min	max
eclipse (DaCapo)	large	10	58	2	3	2	12
hsqldb (DaCapo)	large	400	532	1	79	2	1020
lusearch (DaCapo)	large	64	248	2	13	1	121
xalan (DaCapo)	large	16	404	3	12	1	513
jbb2005 (SPEC)	16 warehouses	16	2371	0	121	0	413
jServer2004 (SPEC)	Tx=20	2848	11130	0	301	0	421
ECPerf	Tx=20	1368	10456	0	271	0	357

Table 1. Contention characteristic of each benchmark. Note that the column "Contention per object" reports the actual occurrences of contention observed in HotSpot.

Opteron processors with 2GB of memory; it runs Fedora Core 2 Linux. The client machine is a single-processor 1.6 GHz Athlon with 1GB of memory.

We executed each application on *default* five times and on *CASe* five times. We then compared the best run from *CASe* to the best run from *default*. It is worth noting that when the best run of *CASe* is compared to the worst run of *CASe* and the best run of *default* to the worst run of *default*, the performance differences range from 0.1% to 12.9% in all benchmarks. The biggest difference (12.9%) occurs when *xalan* is executed using two processors in *CASe*. The biggest difference in *default* (11.1%) occurs when *lusearch* is executed with two processors. The average performance difference when 1, 2, 4, 8, and 16 processors are utilized is 5.1%. Interestingly, as the number of processors becomes larger than two (i.e., utilizing 4, 8, and 16 processors), the average performance difference reduces to 3.3%. In addition, we found the differences in throughput performance of server-side benchmarks to be less than 5%. This is because these benchmarks run for a long time (tens of minutes to a few hours), resulting in more consistent overall performances.

3.1 Benchmark Characteristics

Table 1 describes each benchmark and the possible sources of contention obtained through source code inspection. The numbers of threads generated by these benchmarks vary significantly. In addition, the numbers of shared objects and contended objects also vary significantly. Also note that we report the number of objects that are accessible by multiple threads (shared objects in the fourth column). However, not all of these objects experience contention; therefore, we see that there are shared objects experiencing no contention in *jbb2005*, *jAppServer2004*, and *ECPerf*.

eclipse (DaCapo) is a simplified integrated development environment (IDE) without displaying GUI. It performs a subset of five tasks in IDE (setting up workspace, creating projects, and typing hierarchy tests, AST tests, search tests). In our experiment, we configured *eclipse* to execute the full set of tasks. Since these tasks are independent from each other, they only have a few occurrences of resource contention [7].

hsqldb (DaCapo) is an SQL relational database engine written in Java. Dacapo provides a modified benchmark program JDBCBench to test an *hsqldb* database that emulates a transaction processing system (TPS) [7]. The TPS database has 4 tables: branch, teller and account and history. In our experiments, we ran 400 client threads. Each thread performs 50 transactions. Each transaction randomly chooses an account, teller or branch and updates its balance. We expected that threads, which tend to update the same table, would have more contention than threads updating different tables. One major shortcoming of *hsqldb* is that while the database server is multithreaded, the core database engine is not. Thus, requests are serviced sequentially [17].

lusearch (DaCapo) is a tool that performs a text search of keywords over the Shakespeare Corpus (42 documents) [7]. It creates multiple threads to search these documents in parallel. It also maintains a hit queue, which ranks the documents by the order of keyword hits. Each thread updates the hit queue during search. Contention occurs when multiple threads update the hit queue concurrently. In our experiments, we ran 64 threads.

xalan (DaCapo) is an XSLT processor for transforming XML documents [7]. It pushes some pre-selected files to a work queue and generates several threads to parse these files independently. Once a thread has completed transformation of a XML file, it removes the file from the work queue. Contention occurs when threads update the work queue simultaneously. In our experiments, we modified the number of pre-selected files to 64 and the number of work threads to 16.

jbb2005 (SPEC) is a Java-based benchmark that models a wholesale company, with warehouses that serve a number of districts. Each warehouse is implemented as a simple database by using HashMaps or TreeMaps. Each thread simulates a customer which accesses a fixed warehouse for the lifetime of the experiment [30]. Given the nature of the benchmark, threads accessing the same warehouse should have more contention than threads accessing different warehouses. In our experiments, we used 16 warehouses, resulting in 16 active threads.

jAppServer2004 (SPEC) is a standardized benchmark for testing the performance of Java Application Servers. It emulates an automobile manufacturing company and its associated dealerships [29]. It complies with the J2EE 1.3 specification. We ran it on JBoss[4] and used MySQL[5] as database server. The level of workload can be configured by transaction rate (Tx). This workload stresses the ability of the Web and EJB containers to handle the complexities of memory management, connection pooling, passivation/activation, caching, etc. The throughput of the benchmark is measured in JOPS (job operations per second). We set Tx to 20, which generates 2848 threads. These threads will intensively contend on sockets, cache pools, and databases.

ECPerf was introduced by Sun Microsystems in 2001. It consists of a specification and a benchmarking kit, which is designed specifically to test and measure performance and scalability of J2EE application servers. ECPerf does not provide a workload related to web-tier and Java Message Service (JMS). Similar to *jAppServer2004*, the level of workload can be configured by transaction rate (Tx) [31]. The throughput of the benchmark is measured in JOPS. We set Tx to 20, which generates 1368 threads. Notice that the number of threads created by ECPerf is roughly half of the number of threads created by jAppServer2004 using a similar workload setting. These threads also contend on sockets, cache pools, and databases.

3.2 Performance Tuning

The proposed system introduces several parameters that must be tuned to achieve a balance between good performance and low overhead. In this section, we study the effects of these parameters on performance and clustering accuracy.

Similarity Threshold: As a reminder, *similarity* between two threads is used as a metric to determine if two threads are likely to contend. Its value is a normalized product of two *conVectors*, each of which is a counting vector that indicates the number of monitor entries perform by a particular thread on each shared object. Because the JVM derives *conVectors* from processing *seVectors*, the size of *seVector* (s) can have a profound effect on the clustering accuracy and runtime overhead.

First, we investigated the effects of changing the values of s and *simThreshold* on the overhead to record synchronization events, form clusters, and generate a thread-to-processor mapping plan. We conducted our study on three benchmarks representing three distinct workloads and contention behaviors: *eclipse* (representative of desktop application with 10 threads and very few occurrences of contention), *jbb2005* (representative of small server application with 16 threads and moderate contention), and *jAppServer2004* (representative of large server applications with thousands of threads and heavy contention).

Figure 3 presents the execution overhead inside the JVM (i.e. overhead for clustering and generating thread-to-CPU mapping plan) over a wide range of *seVector* size (s) and simThreshold. Note that the overall overhead (i.e., overheads of JVM and kernel) is presented in Section 3.3. We only used four *simThreshold* values (i.e., 0.1, 0.2, 0.3, and 0.4) because our experimental result reveals that the *similarity* between two threads rarely exceeds 0.45.

The graph shows an expected trend of increasing overhead with larger s. Our experiment shows that the overhead is generally less than 2% of the overall execution time when s is less than or equal to 2048. Note that in *eclipse*, the overhead no longer increases when s is greater than 2048. This is because the total number of monitor_entry events of each thread in *eclipse* is less than 2048. Also, it is worth noticing that varying *simThreshold* has very little effect on the overhead in *eclipse*, but has significant effects on *jbb2005* and *jAppServer2004*. In subsequent experiments, we set s to 2048 based on the results of this study.

Another observation is that smaller *simThreshold* yields less computational overhead under a fixed value of s. With smaller *simThreshold*, threads are more likely to be clustered. Therefore, fewer clusters will be formed. Since the number of clusters determines the overhead of the processor assignment, fewer clusters result in smaller overhead.

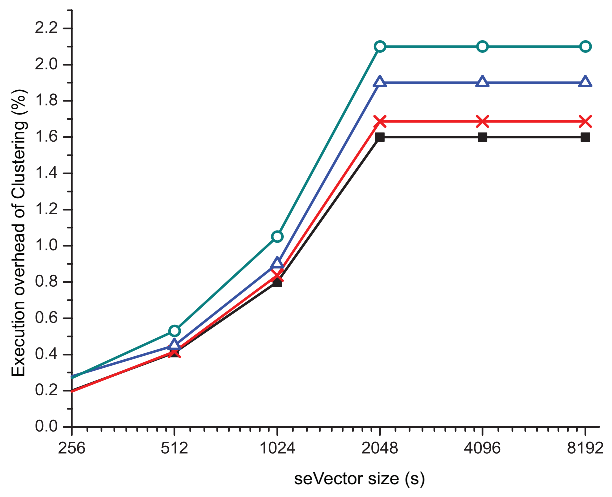
Next, we investigated the effects of varying the values of s and *simThreshold* on clustering accuracy. To conduct this investigation, we created an oracle for each benchmark by manually forming clusters using a combination of a brute-force approach and a source code investigation to establish the contention characteristics. By inspecting the source code, we discovered that we can cluster threads based on task in *eclipse*. Therefore, threads that are created to perform a certain task are assigned to the same cluster. For *jbb2005*, we observed that threads participating in a transaction also share resources. Thus, we clustered threads based on the frequency with which they accessed each warehouse. For *jAppServer2004* and *ECPerf*, we observed that threads with the same functionality share a large number of resources; thus, they are clustered together.

Next, we built a relationship matrix, M of the manually-formed clusters. M is a matrix of size n where n is number of threads. M is generated as follows: $M[i][j]$ is 1 (1⇐ i,j ⇐ n) if thread i and j are in the same cluster, indicating an inclusive relationship. Otherwise, $M[i][j]$ is 0, denoting an exclusive relationship.

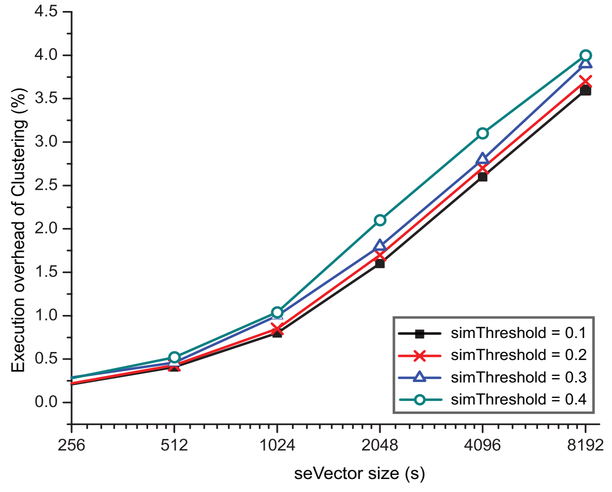
In a similar fashion, we constructed another relationship matrix, M_1, that is based on the clustering result of our proposed algorithm. Once the two matrices were constructed, we calculated the accuracy of our clustering algorithm using the following formula:

[4] See www.jboss.org for more information.

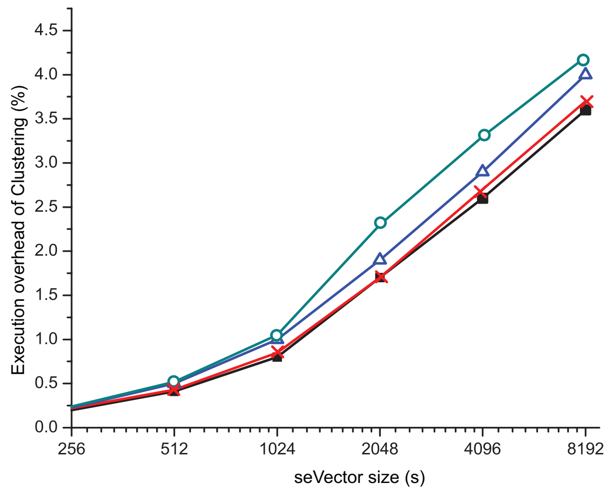
[5] See www.mysql.com for more information.



(a) *eclipse*



(b) *jbb2005*



(c) *jAppServer2004*

Figure 3. Analysis of runtime overhead of CA-Scheduler inside the JVM

$$accuracy = \frac{1}{C_n^2} \sum_{i=1}^{n} \sum_{j=i+1}^{n} A_{ij},$$

$$where\ A_{ij} = \begin{cases} 1, & when \quad M[i][j] = M_1[i][j] \\ 0, & when \quad M[i][j] \neq M_1[i][j] \end{cases}$$

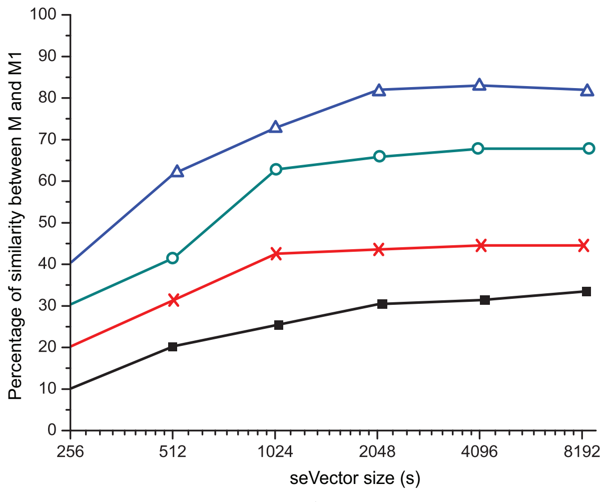
A_{ij} indicates whether the relationship (i.e., inclusive or exclusive) between thread i and j is the same in M and M_1. C_n^2 refers to the total number of thread pairs, which is equal to $n \times \frac{(n-1)}{2}$. Therefore, $accuracy$ indicates whether M_1 is similar to M. In this scheme, the maximum accuracy is 1, meaning that the two matrices, M and M_1 are exactly the same. Figure 4 shows the accuracy over wide-ranging values of s and $simThreshold$.

ciThreshold: In this experiment, the values of *s* and *simThreshold* are set to 2048 and 0.3, respectively. As a reminder, *ciThreshold* is used to determine whether a cluster should be classified as strong-contention or weak-contention. Thus, it can have a profound effect on our system's ability to perform load balancing. If we set *ciThreshold* to 1.0, all clusters are classified as weak-contention, meaning that every cluster can be split to achieve a good balance. However, such action can result in a much higher number of inter-processor contention. On the other hand, if we set *ciThreshold* to 0, all clusters are categorized as strong-contention, resulting in less effective load balancing. Figure 6 depicts the effect of varying *ciThreshold* values on the overall performance and contention.

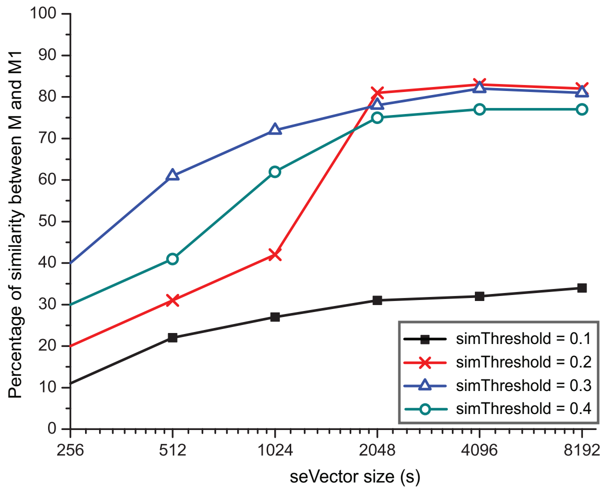
3.3 Discussion

The results of our study indicate that a good balance between high clustering accuracy (70% to 80% in all three benchmarks) and low clustering overhead (less than 2% in all three applications) can be achieved if we set s to 2048 and simThreshold to 0.3. Our study of *ciThreshold* also shows that carefully selecting the value of *ciThreshold* can improve the overall performance and reduce contention. The results also reveal that the optimal values of *ciThreshold* can vary across applications. For example, the value of *ciThreshold* that allows the CA-Scheduler to yield its highest performance and reduce the most contention is 0.4 for *eclipse* and *jAppServer2004* and 0.2 for *jbb2005*. However, the value of 0.4 is a good compromise, as it still results in a very good performance improvement for *jbb2005*, but a slightly lower contention reduction than the optimal value.

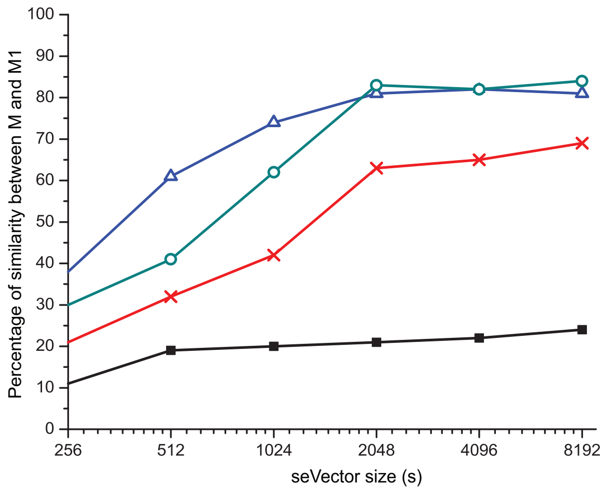
We then conducted an analysis of the overall execution overhead. We set the values of s to 2048, *simThreshold* to 0.3, and *ciThreshold* to 0.4. As shown in Figure 5, the CA-Scheduler only incurred a maximum of 3.5% execution overhead (*lusearch*). The investigation also reveals that recoding synchronization events and building *conVectors* account for 83% to 92% of the entire overhead.



(a) *eclipse*



(b) *jbb2005*



(c) *jAppServer2004*

Figure 4. Accuracy analysis of the clustering algorithm employed in CA-Scheduler

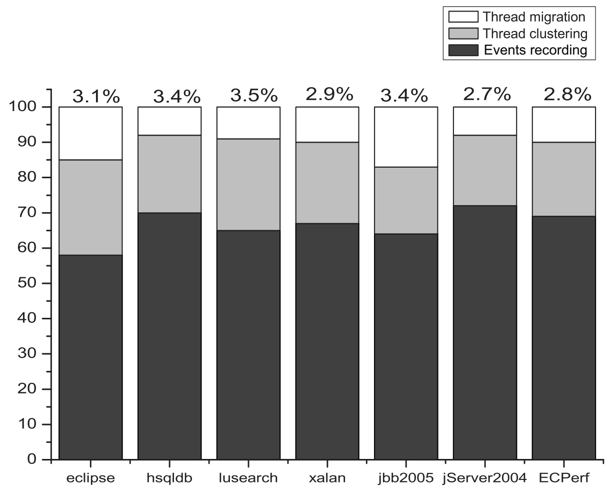


Figure 5. Overall runtime overhead of CA-Scheduler

4. Performance Evaluation

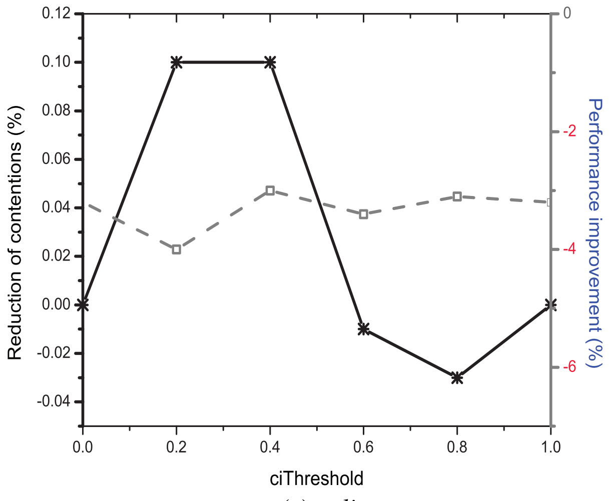
In this section, we investigate the effect of the CA-Scheduler on reducing the number of contention and improving overall performance. We also compare the CPU-scalability of *CASe* with that of *default*. We set s to 2048, *simThreshold* to 0.3, and *ciThreshold* to 0.4.

4.1 Contention Reduction

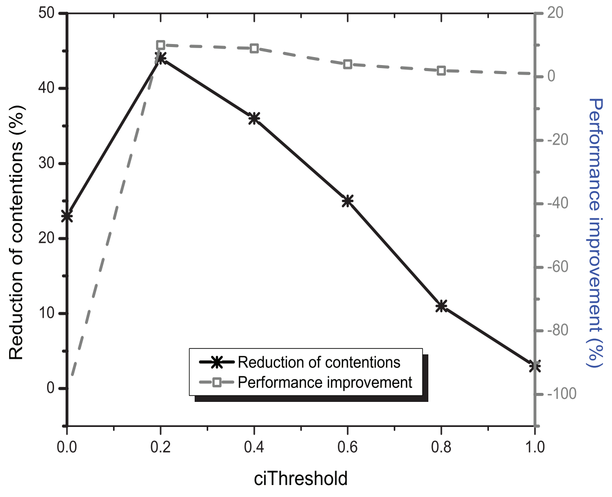
In this study, we compare the number of contention between *CASe* and *default*. Furthermore, we create two versions of CASe: Version 1 includes clustering to address only interprocessor contention, and Version 2 includes clustering and Critical Section First (CSF) scheduling to address both interprocessor- and intraprocessor-contention.

CASe-V1 can reasonably reduce the number of contention in four out of seven applications. We can achieve these reductions because our approach assigns threads, which contend heavily, to the same processor. This assignment reduces the number of inter-processor contention (ranging from 10% to 29%). With the CSF scheduling strategy (CASe-V2), our approach can also reduce intraprocessor contention. As a result, the overall reductions range from 15% to 45%. This result indicates that intraprocessor contention is a significant factor when the number of threads exceeds the number of processors (400 threads in *hsqldb* to nearly 3000 threads in *jAppServer2004*).

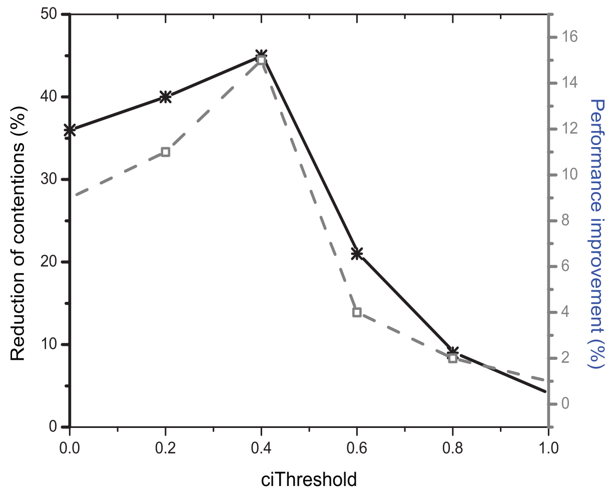
For the remaining three applications, our approach achieves a very small reduction because contention among threads in these applications is rare. For example, *xalan* only uses two synchronized methods to update the work queue (adding or removing an XML transformation task). The execution time of these methods is also very short. Therefore, contention rarely occurs. As a result, our approach only reduces the number of contention by 3%.



(a) *eclipse*



(b) *jbb2005*



(c) *jAppServer2004*

Figure 6. Investigating the effects of *ciThreshold* on performance

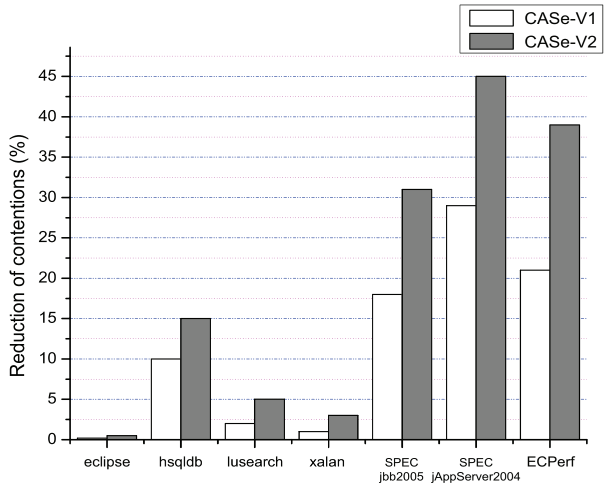


Figure 7. Contention reductions over *default*

4.2 Overall Performance Improvement

In *eclipse*, *lusearch*, *hsqldb* and *xalan*, performance is measured by execution time. In *jbb2005*, *jAppServer2004* and *ECPerf*, performance is measured in jobs per second (JOPS). Figure 8 shows the performance improvements of CASe-V1 and CASe-V2 over *default*.

As shown in Figure 8, *eclipse* suffers a 3% degradation in performance. This is because our approach can achieve a very small contention reduction in *eclipse*. Therefore, the degradation of 3% is mainly caused by the overhead incurred by the CA-Scheduler. In *lusearch* and *xalan*, we do not achieve any noticeable performance improvement. Again, this is due to small numbers of contention in these two benchmarks.

For the remaining four benchmarks, performance improvements of CASe-V2 range from 7% to 15%. The magnitude of performance gain reflects its ability to reduce the number of contention. For example, in *jAppServer2004*, CASe-V2 is able to improve performance by 15% by eliminating 45% of the contention. In addition, CSF scheduling decreases the time that a thread must spend waiting to enter critical sections.

So far, we have measured execution times and throughput performances using all 16 processors. Next, we investigate the CA-Schedulers ability to perform with a varying number of processors. We report performance improvement in terms of *speedup* with respect to the performance of *default* with one core [22]. For example, the speedup of the CA-Scheduler with 16-core in *jAppServer2004* is $\frac{throughput_{CA-16}}{throughput_{default-1}}$, where $throughput_{CA-16}$ represents the throughput performance of *jAppServer2004* running on CASe-V2 with 16 processors and $throughput_{default-1}$ represents the throughput performance of *jAppServer2004* running on *default* with one processor core.

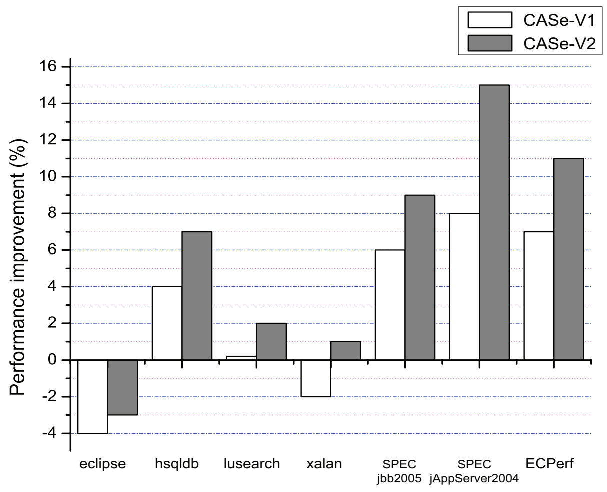


Figure 8. Performance improvements over *default*

We maintained the same workload while increasing the number of CPU cores. After we added a CPU, we measured the performance of each application. As shown in Figure 9, CASe-V2 can perform worse than *default* when the number of cores is small. This is because a smaller number of cores reduces contention. Although CASe-V2 performs worse with one core, its performance improves quickly and surpasses that of *default*, as the number of cores increases in five out of seven applications. Two exceptions are *eclipse* and *xalan*. In these two benchmarks, the CA-Scheduler is ineffective due to a small number of contention.

In *jbb2005*, the maximum throughput of *default* is achieved (58839 JOPS) when all 16 cores are used. CASe-V2 can achieve a similar throughput by using only 12 cores. Because this application has only 16 active threads, there is a one-to-one mapping between threads and CPUs, yielding maximum parallelism. This result shows that CASe-V2 utilizes CPUs much more efficiently than *default* in *jbb2005*. This preliminary result indicates that it is possible to achieve better CPU utilization by favoring contention reduction over increasing CPU cores.

4.3 CPU Scalability

In this section, we compare the CPU-scalability of the CA-Scheduler (using CASe-V2) with *default*. Note that CPU-scalability refers to an increase in performance (i.e., an increase of throughput or a decrease in execution time) with respect to an increase in the number of CPU cores.

As shown in Figure 9, the speedups of most benchmarks begin to saturate after the number of CPU cores reaches a certain number. For example, in *eclipse*, peak performance is achieved at five or more CPUs. This is because there are five independent tasks (10 threads in total), which seldom contend. Therefore, there is no significant performance gain after five CPU cores. In *jbb2005*, the throughput begins to saturate at 12 cores. Between 12 cores to 16 cores, the throughput improvement is less dramatic.

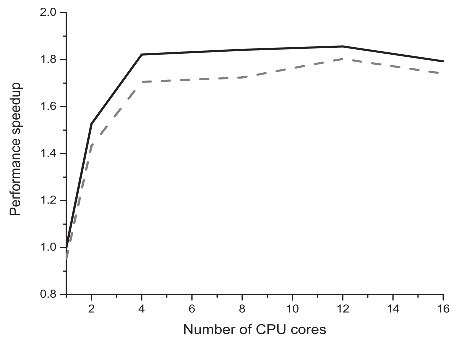
As can be seen in Figure 9, the CA-Scheduler improves CPU scalability in five out of seven benchmarks. We also discovered an interesting scalability behavior in *hsqldb*, *jAppServer2004*, and *ECPerf* when *default* is used. In these three applications, performance actually degrades when more CPUs are added to the system. In *hsqldb*, a smaller number of CPUs limits the number of requests the system can take at one time. As we add more CPUs, the additional computation power is used mainly to take more requests. This does not help with servicing queries because the core database engine is single-threaded [17] and must share the processors with the rest of the threads. Thus, we see reduced performance in the *default* system when the number of processors is 12 or more.

On the other hand, CASe-V2 assigns a higher priority to threads in critical sections. Because the core engine handles requests in a serialized fashion, CASe-V2 schedules the core engine more frequently than other threads. This means that in a single core system, threads taking requests are less frequently scheduled. As we add more CPUs to the system, one processor is mostly used by the core engine, and the remaining processors are used to accept concurrent requests and queue them up for the core engine. As a result, CASe-V2 scales well with more processor cores and eventually outperforms the default system once the number of CPUs is 14 or more.

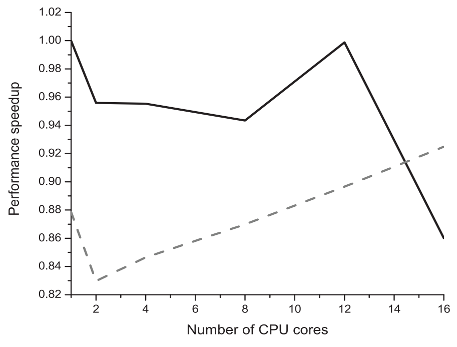
In *jAppServer2004* and *ECPerf*, CASe-V2 shows modest improvements in the scalability over *default*, which slightly degrades performance with more CPU cores. Because the numbers of threads in these two applications are very large (several hundred concurrent threads with thousands created throughout execution), increasing the number of processors from 1 to 16 modestly improves parallelism in these two applications. However, it appears that reducing contention, as in the case of CASe-V2, allows the system to scale in spite of a modest increase in parallelism.

5. Other Considerations

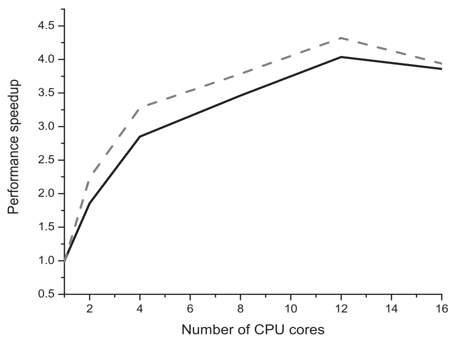
Security: In our current implementation, we have not incorporated any security measures to proactively prevent other threads from registering with the kernel. Therefore, it is possible for native threads, as an example, to register with the kernel and to be scheduled based on the contention-aware policy. However, it is not possible for unregistered threads to simply invoke any kind of system call to renew execution quanta, as the renewal process is done automatically by the kernel. One possible security measure is to make the registration process more secure (e.g., encrypted policy configuration files, authentication of all processes trying to register, and tighter control from system administrators).



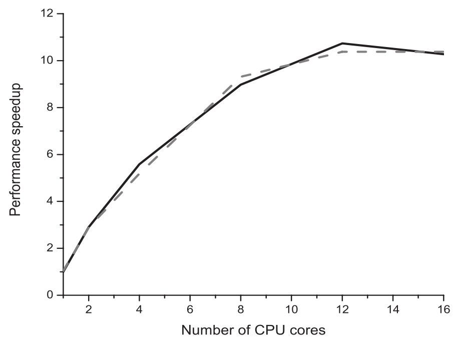
(a) *eclipse*



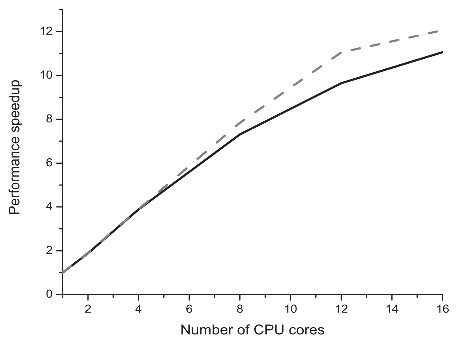
(b) *hsqldb*



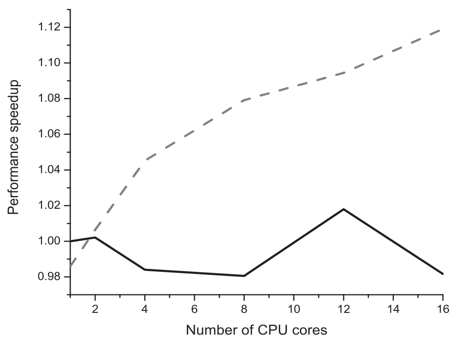
(c) *lusearch*



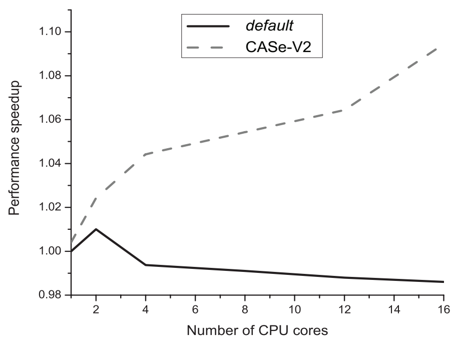
(d) *xalan*



(e) *jbb2005*



(f) *jAppServer2004*



(g) *ECPerf*

Figure 9. Comparing Performance and CPU-scalability of CASe-V2 with *default*

Incremental Migration: The current implementation of the thread migration mechanism assigns thread-to-CPU in a stop-the-world fashion, meaning that all Java threads must stop executing during this process. Because migration seldom occurs, its overhead appears to be small when it is averaged over the entire execution of a program. However, we found that each migration causes the application server benchmarks to exhibit noticeable pauses of about 500 milliseconds to 1 second. As part of future work, we plan to make the migration process incremental to reduce the length of each pause. We expect that such an approach will make pauses less noticeable. However, it will also result in higher migration overhead due to additional bookkeeping activities and longer delays before the approach can reap its full benefits.

Portability: It is also possible to implement our scheduler as a user-level thread management package. One approach is to map a cluster to a kernel thread and then rely on the user-level thread manager to multiplex threads belonging to a cluster. However, it is likely that such an approach would result in a longer thread waiting time. For example, if three actual time quanta are needed for a thread to complete executing in a critical section, in our current approach, the thread can get through this critical section in one turn on the CPU (2 renewals). In the user-level thread approach, the thread may be suspended by the kernel before it can get through the critical section, resulting in more convoying and longer waiting time. On the other hand, the user-level thread approach is more portable than the kernel-level thread approach.

Supporting Native Threads: Currently, our prototype only works with Java. However, integrating the proposed clustering and planning mechanism to languages such as C and C++ should be quite straightforward. For example, similar clustering and thread-to-CPU mapping mechanisms can be included in threading libraries such as pthread.

6. Related Work

The goal of this work is very similar to many other works that attempt to make thread-level parallelism more efficient in multiprocessor systems. Our approach is unique because it achieves the same goal through operating system augmentation instead of (i) relying on programmers to provide lock-free code (e.g., lock-free data structures [24]); (ii) employing runtime systems to provide lock-free execution (e.g., hardware and software transactional memory [16, 14]); and (iii) utilizing hardware to assist with lock-free execution (e.g. speculative lock elision [23]). One common characteristic of these techniques is that they treat scheduling decisions made by the kernels as "a fact of life." Therefore, these three approaches are designed to "cope" with contention that results from these scheduling decisions. Our approach differs from these three techniques in that it first attempts to proactively improve the scheduling, so that the execution sequence experiences few occurrences of lock contention. Any unavoidable contention is then managed using existing approaches. Thus, our proposed work is orthogonal but complementary to these three techniques.

In addition, recent emergence of transactional memory also promises less effort by programmers to coordinate parallelism [13]. Currently, most transactional memory systems are tuned according to the conventional wisdom that conflicts rarely occur, and thus aborting memory transactions, a typically more expensive operation, should rarely take place [16, 21]. However, this wisdom has not been verified in large multithreaded applications such as Java application servers with hundreds to thousands of concurrently running threads manipulating unstructured data (e.g. lists, trees, graphs, hash tables). It is very likely that applications wanting to utilize transactional memory have to be partially rewritten, or at the very least, recompiled [8]. Thus, such restrictions limit the applicability of transactional memory in currently deployed software. As shown in this paper, our proposed solution can work well with large application servers. Moreover, our solution allows existing applications to immediately take advantage of its benefits without rewriting the application code.

There have been several research efforts that attempt to reduce contention through more favorable scheduling. Work by Tucker and Gupta suggests that one way to prevent early preemption of processes in critical sections is to make the number of runnable processes the same as the number of processors. This effectively eliminates the need for preemption [33]. Because they only evaluated their work using highly parallel applications, it is unclear how such an approach would perform in large multithreaded applications with a high frequency of contention.

Work by Anderson et al. [2] proposed scheduler activations to address the issue that a user-level thread is blocked or preempted when it is in critical section. They adopted a solution based on recovery. That is, if a thread has been preempted in a critical section, the thread is allowed to temporarily continue until it exits the critical section. Marsh et al. [19] proposed a set of kernel mechanisms and conventions to grant first-class to user-level threads. Their scheme provided coordination between scheduling and synchronization to reduce waiting time caused by lock contention. Similarly, our work employs a time-slice renewing mechanism to reduce contention. However, our approach also employs clustering and dynamic prioritization to further reduce waiting time.

Work by Tucker et al. [34] introduces a scheduling control mechanism to provide "limited control over the scheduling of a lightweight process" in Solaris operating system. Function *schedctl_start* is used to provide a hint to the kernel that preemption of a lightweight process should be avoided. Function *schedctl_stop* is used to remove the hint. One common use of this mechanism is to "block preemption" while a thread is holding a lock. Because we implemented our mechanism in Linux, such feature is not available to us. However, we can see that the availability of such feature can support the implementation of the proposed CSF scheduling policy by utilizing blocking preemption instead of utilizing our current technique of increasing execution priority.

Rossbach et al. [25] presented a variant of Linux called *txLinux* to support hardware transactional memory. TxLinux provided a transaction-aware scheduler that takes advantage of processes' transaction states to mitigate the effects of high contention. The scheduler makes scheduling decisions or assigns priority based on the current transaction state. Similarly, our approach prioritizes threads based on their lock usage.

Tam et al. [32] proposed a clustered scheduling scheme to reduce high-latency due to cross-chip sharing. They used a similar approach to cluster threads that heavily share data. Their work leveraged information available in the hardware performance monitoring to guide thread clustering to reduce the cost of inter-chip cache snooping in chip-multiprocessor systems. On the other hand, our work exploits runtime information from high-level language virtual machines to guide clustering and reduce lock contention. With the CSF scheduler, our work also reduces the number of intra-processor contention.

7. Conclusions

Lock contention is a major bottleneck that not only affects performance, but can also affect scalability of multithreaded Java applications. To reduce the occurrences of lock contention, we introduced a Contention-Aware scheduler (CA-Scheduler) that maps threads sharing the same lock-protected resources to the same processor.

In addition, we introduced a critical-section-first scheduling strategy, which considers lock usage as a scheduling criterion to further reduce thread waiting time due to lock contention. Our experimental results show that the CA-Scheduler can achieve significant performance improvement in applications with heavy lock-contention (up to 15%) without incurring significant runtime overhead (3.5% of overall execution time).

8. Acknowledgments

This work was supported in part by the National Science Foundation under Award Numbers CNS-0411043 and CNS-0720757. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We thank the anonymous reviewers for providing valuable comments and insights that significantly improve the final version of this paper. This work was completed utilizing the Research Computing Facility with the associated USCMS Tier-2 site at the University of Nebraska – Lincoln.

References

[1] J. Aas. Understanding the Linux 2.6.8.1 Scheduler. On-line article, 2006. http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 95–109, New York, NY, 1991.

[3] M. Arnold, A. Welc, and V. T. Rajan. Improving Virtual Machine Performance Using a Cross-Run Profile Repository. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA)*, pages 297–311, San Diego, CA, 2005.

[4] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 258–268, Montreal, Quebec, Canada, June 1998.

[5] J. C. Bezdek, R. Ehrlich, and W. Full. FCM: The fuzzy C-Means Clustering Algorithm. *Computers & Geosciences*, 10(2-3):191–203, 1984.

[6] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, November 1995.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 169–190, Portland, OR, 2006.

[8] B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Cao Minh, L. Hammond, C. Kozyrakis, and K. and Olukotun. Transactional Execution of Java Programs. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. Oct 2005.

[9] M. Cohen, S. B. Kooi, and W. Srisa-an. Clustering the Heap in Multi-Threaded Applications for Improved Garbage Collection. In *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1901–1908, Seattle, WA, 2006.

[10] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 15–24, San Francisco, CA, 2003.

[11] R. Dimpsey, R. Arora, and K. Kuiper. Java Server Performance: A Case Study of Building Efficient, Scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.

[12] C. Grzegorczyk, S. Soman, C. Krintz, and R. Wolski. Isla Vista Heap Sizing: Using Feedback to Avoid Paging. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 325–340, San Jose, CA, March 2007.

[13] T. Harris, A. Cristal, O. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional Memory: An Overview. *IEEE Micro*, 27(3):8–29, May-June 2007.

[14] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 14–25. Ottawa, Ontario, Canada, Jun 2006.

[15] J. A. Hartigan and M. A. Wong. A K-Means Clustering Algorithm. *Applied Statistics*, 28:100–108, 1979.

[16] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 289–300. May 1993.

[17] HSQL Database Engine. hsqldb. On-Line Documentation, Last visited: December 2007. http://hsqldb.org/web/hsqlFAQ.html.

[18] IBM. Jikes RVM. http://jikesrvm.sourceforge.net.

[19] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-Class User-Level Threads. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 110–121, New York, NY, 1991.

[20] Microsoft Corp. Using Microsoft Virtual PC 2007 for Application Compatibility. White Paper, August 2006. http://www.microsoft.com/windows/products/winfamily/virtualpc/appcompat.mspx.

[21] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 254–265. Feb 2006.

[22] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design (3rd ed.): the Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2004.

[23] R. Rajwar and J. R. Goodman. Speculaive Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 294–305, Austin, TX, 2001.

[24] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 5–17, San Jose, CA, 2002.

[25] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–102, New York, NY, 2007.

[26] Silberschatz and Galvin and Gagne. *Operating System Concepts, 7th Edition*. Addison Wesley, 2007.

[27] J. Singer, G. Brown, I. Watson, and J. Cavazos. Intelligent Selection of Application-Specific Garbage Collectors. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 91–102, Montréal, Quebec, Canada, 2007.

[28] S. Soman, C. Krintz, and D. F. Bacon. Dynamic Selection of Application-Specific Garbage Collectors. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 49–60, Vancouver, BC, Canada, 2004.

[29] Standard Performance Evaluation Corporation. SPECjAppServer2004 user's guide. http://www.spec.org.

[30] Standard Performance Evaluation Corporation. SPECjbb2005. On-Line Documentation, Last visited: July 2007. http://www.spec.org/jbb2005.

[31] Sun Microsystems. ECPERF. http://java.sun.com/developer/earlyAccess/j2ee/ecperf/download.html.

[32] D. Tam, R. Azimi, and M. Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. *SIGOPS Operating System Review*, 41(3):47–58, 2007.

[33] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 159–166, New York, NY, 1989.

[34] A. Tucker, B. Smaalders, D. Singleton, and N. Kosche. US patent 5,937,187: Method and Apparatus for Execution and Preemption Control of Computer Process Entities, 1999.

[35] V. Uhlig. The Mechanics of In-Kernel Synchronization for a Scalable Microkernel. *SIGOPS Operating System Review*, 41(4):49–58, 2007.

[36] F. Xian, W. Srisa-an, and H. Jiang. Allocation-Phase Aware Thread Scheduling Policies to Improve Garbage Collection Performance. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 79–90, Montréal, Quebec, Canada, October 2007.

[37] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual Memory Support for Garbage-Collected Applications. In *Proceedings of the USENIX Conference on Operating System Design and Implementation (OSDI)*, pages 103–116, Seattle, WA, November 2006.