# Multithreading in Java: Performance and Scalability on Multicore Systems

Kuo-Yi Chen, *Student Member*, *IEEE*,
J. Morris Chang, *Senior Member*, *IEEE*, and Ting-Wei Hou, *Member*, *IEEE*

**Abstract**—The performance and scalability issues of multithreaded Java programs on multicore systems are studied in this paper. First, we examine the performance scaling of benchmarks with various numbers of processor cores and application threads. Second, by correlating low-level hardware performance data to JVM threads and system components, the detail analyses of performance and scalability are presented, such as the hardware stall events and memory system latencies. Third, the usages of memory resource are detailed to observe the potential bottlenecks. Finally, the JVM tuning techniques are proposed to alleviate the bottlenecks, and improve the performance and scalability. Several key findings are revealed through this study. First, the lock contentions usually lead to a strong limitation of scalability. Second, in terms of memory access latencies, the most of memory stalls are produced by L2 cache misses and cache-to-cache transfers. Finally, the overhead of minor garbage collections could be an important factor of throughput reductions. Based on these findings, the appropriate Java Virtual Machine (JVM) tuning techniques are examined in this study. We observe that the use of a parallel garbage collector and an appropriate ratio of young to old generation can alleviate the overhead of minor collection and improve the efficiency of garbage collections. Moreover, the cache utilizations could be enhanced with the use of thread-local allocation buffer, and then leads to the performance improvements significantly.

**Index Terms**—Garbage collection, Java, lock contention, multicore, performance counter, scalability, virtual machine.

## 1 Introduction

In recent years, Java language has become a popular choice for development of multithreaded applications and long-running servers [1]. Designed with many advanced features, such as automatic memory management, cross-platform portability, and enforced security check, Java is also an excellent programming language used on various platforms. The multithreading feature of Java is also employed on platforms, largely on the server systems, to gain higher throughput and faster response time.

Today, some of the most common multithreaded Java applications in a server environment are: IBM WebSphere, BEA WebLogic, JBoss, Oracle Application Server, and Sun Java System Application Server. These applications are highly threaded and memory-intensive and tend to be operated on multiprocessor platforms, such as multicore systems. Due to the advance of semiconductor technology, multicore systems are becoming affordable, accordingly driving the widespread deployment of these applications. However, the performance and scalability of multithreaded Java applications running on multicore systems remain a major concern and were rarely reported.

Java employs sophisticated runtime environment, Java Virtual Machine (*JVM*), to obtain productivity, portability, and security. The versatility brought by this extra layer comes with the high cost of performance degradation. Therefore, the performance analysis becomes very important. However, to characterize the performance of multithreaded Java applications remains quite challenging. The challenge is due to that performance of a system should be observed by multiple levels, such as hardware, operating systems, runtime environments, and applications. Moreover, the complexity of a multithreaded application increases the difficulty of characterization.

As the use of multicore processors is becoming a common commodity, software developers are investing heavily in developing multithreaded applications. However, ensuring acceptable performance of these applications on the myriad of parallel platform is hard to reach. The disparity between application's demand and available resources decreases the application performance. On one side, if the application demand is represented by its degree of parallelism and required resources exceed the available processor capability, the system will exhibit low scalability. On the other side, low application demand means that the system might be under-utilized.

Due to the complexity, performance issues regarding multithreading in Java is conspicuously hard to deal with. Challenge is being faced in achieving higher performance as multithreaded Java applications scale with multicore systems. Intuitively, the resource contention and memory system latencies are the major performance bottlenecks. Therefore, the bottleneck analyses could be paramount for the improvements of performance and scalability.

The goal of this paper is to study the performance and scalability issues of multithreaded Java applications on multicore systems. The potential bottlenecks of scalability and performance are identified by the thorough study of scaling of Java on multicore systems, and then appropriate

- *K.-Y. Chen and T.-W. Hou are with the Department of Engineering Science, National Cheng Kung University, Tainan, Taiwan. E-mail: Kuoyichen@iastate.edu, houtw@mail.ncku.edu.tw.*
- *J.M. Chang is with the Electrical Engineering Department, Iowa State University, Ames, IA 50010. E-mail: morris@iastate.edu.*

*Manuscript received 26 Apr. 2009; revised 31 Dec. 2009; accepted 12 July 2010; published online 30 Nov. 2010.*
*Recommended for acceptance by L. John.*
*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-04-0178.*
*Digital Object Identifier no. 10.1109/TC.2010.232.*

JVM tuning techniques are examined. In order to accomplish this goal, the following steps of investigation are performed in this paper.

- Performance measurements

  Eight widely used multithreaded Java benchmarks are used to execute with a state-of-the-art JVM on an eight cores system. All benchmarks execute with various numbers of cores and a portion of benchmarks execute with various numbers of application threads. Thus the detailed information of the throughput could be observed for the further performance analyses.
- Bottlenecks identification

  In order to detect bottlenecks of performance and scalability at multiple levels, the low-level hardware performance data is correlated to the high-level software behavior. Thus the CPU cycle usage could be detailed, and then bottlenecks could be identified.
- JVM tuning techniques

  Once the bottlenecks are identified, appropriate JVM tuning techniques will be discussed for their abilities to reduce the impacts of these bottlenecks.

Through performing these steps of investigation, this paper makes the following contributions. First, the comprehensive characterization of eight widely used multithreaded Java benchmarks is provided. It evaluates the benchmark performance and examines the scalability by varying the number of cores and application threads.

Second, the thorough analyses by breaking down total CPU cycles are provided. The analyses are based on the memory system latency components, types of JVM threads and stall cycles. Thus potential performance bottlenecks could be identified.

Third, two major bottlenecks: the resource contention and memory system latencies are studied. Specifically, we suggest that more attentions should be paid to the lock contentions and L2 cache performance when doing performance improvements for memory-intensive applications in a highly threaded environment.

Finally, the useful findings are revealed by examining tuning techniques. The performance improvements by the use of parallel garbage collector could be observed in a memory-intensive application. The appropriate ratio of new to old generation could reduce the overhand of minor collection. Furthermore, the use of a thread-local heap could improve the cache performance when a memory-intensive Java application runs on multicore systems, although its original intension is to reduce the heap contention [2]. The use of allocation buffer with an appropriate size, between 16 KB and 256 KB, usually leads to performance improvements in our study.

## 2 Methodology

In this section, the experimental environments, including multithreaded benchmarks, and methodologies of implementation and measurement, are detailed as follows:

### 2.1 Experiment Setup

The experiments are based on a server with two Xeon X5355 processors, the Quad-Core Intel CPU, and 16 GB main memory. Fedora core 8 with kernel version 2.6.24 is used as an OS. In order to avoid miscellaneous influence skewing the results, the single user mode of OS is used.

The experiments are performed with a state-of-the-art JVM, the *HotSpot* of *OpenJDK* [3]. The newest version, OpenJDK 1.7, is built for experiments. The HotSpot JVM executes in server mode as server mode instructs the JVM to perform more extensive runtime optimization.

### 2.2 Multithreaded Java Benchmarks

In this study, multithreaded benchmarks, *Eclipse*, *Hsqldb*, *Lusearch*, and *Xalan*, are selected from DaCapo benchmarks [7]. The numbers of threads are predefined and unchangeable in DaCapo, they are nine in Eclipse, 20 in Hsqldb, 32 in Lusearch and 8 in Xalan. The performance can be observed with various numbers of cores, and then scalabilities could be obtained.

In addition, the SPECjbb2005 [8] and JGF multithreaded benchmarks are used to observe the correlation of scalability and various numbers of application threads. The JGF multithreaded benchmarks, *MolDyn*, *MonteCarlo*, and *RayTracer*, are developed by Java Grande Forum. The numbers of application threads is configurable in SPECjbb2005 and JGF multithreaded benchmarks. Thus the variations of scalability could be observed by varying the numbers of application threads.

In order to maintain the constant memory pressure upon each benchmark, we fixed the heap sizes to the three times of minimum heap size of each benchmark for all experiments based on common practices in related studies [6], [11].

### 2.3 Implementation and Measurement

There are many factors, such as retired instructions, memory access latency, access latency, processor utilization, thread synchronization, that could affect the performance. In order to identify the bottlenecks of performance and scalability, it is necessary to obtain the performance data during the execution for further analyses.

In order to obtain the performance data, the *performance counter* is used as a monitor to record the hardware events. In this study, the Intel's *VTune* performance analyzer [4] is used as a performance counter [9]. The performance data can be recorded on a per-thread basis. Thus the unique behavior of each type of JVM threads could be observed. In order to observe the performance variations of benchmarks, the benchmarks are executed with various configurations as follows:

- **Various number of processor cores**

  For each run of a benchmark, the different number of cores from one to eight (one, two, four, six, and eight) is assigned. The use of a single core, which is not a multicore configuration, is included only for comparison.
- **Various number of application threads**

  For each run of SPECjbb2005 and JGF multithreaded benchmarks, the different number of application threads is assigned from one to eight (one, two, four, six, and eight). The use of single application thread is for comparison purpose.

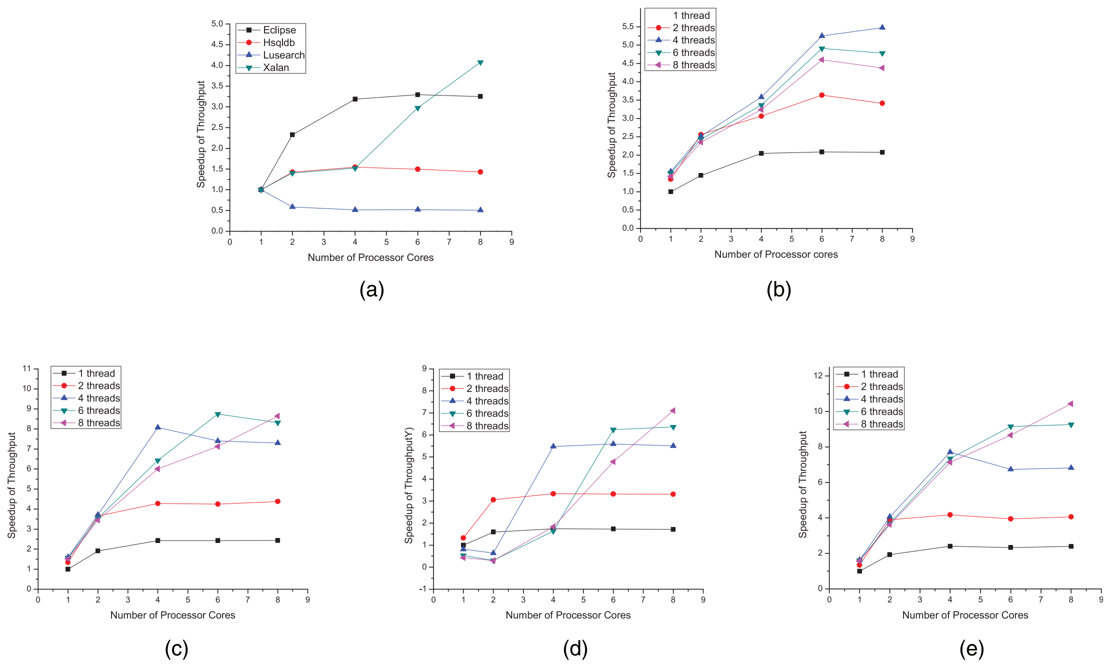


Fig. 1. Throughput Speedup of multithreaded benchmarks. (a) DaCapo benchmarks, (b) SPECjbb2005, (c) MolDyn, (d) MonteCarlo, (e) RayTracer.

- **The use of JVM tuning techniques**
  In order to observe the performance variations of JVM tuning techniques, three techniques are applied to examine their performance. First, the various sizes of a thread-local allocation buffer are applied to examine the impact on memory systems. The results would be detailed in Section 3.7.1. Second, the parallel garbage collector, which is also referred to as the throughput collector [14], is used to compared with the use of default concurrent garbage collector. The results would be detailed in Section 3.7.2. Finally, a tuning technique, which is called *NewRatio* in HotSpot, is examined its improvement of minor garbage collections. The results will be detailed in Section 3.7.3.

In order to perform the comparison of different JVMs, we also apply tuning techniques on Jikes RVM. The comparisons are shown in Section 3.7.4.

## 3 Experimental Results

The experimental results are summarized in this section. The objective is to correlate low-level hardware events to memory components and types of JVM threads. The correlations could be used to identify bottlenecks at multiple levels, and then the strategies for performance enhancement could be derived.

### 3.1 The Types of JVM Threads

When a multithreaded Java benchmark runs with HotSpot, a certain number of JVM threads will be created. In particular, there are six types of JVM threads have been implemented in HotSpot. They are *vm_thread*, *cgc_thread*, *pgc_thread*, *java_thread*, *compiler_thread*, and *watcher_thread*. The character of each JVM thread is shown as follows:

Only one vm_thread is created when a Java application runs with HotSpot. The vm_thread waits the operations which appear in the VMOperationQueue, and then executes operations. Once a *safepoint* has been requested, the vm_thread must wait until all threads are known to be in a safepoint-safe state before executing any operation. A thread lock is used to block all threads during a safepoint. Thus the use of more safepoints could lead to more idle cycles and performance reductions. Thus the number of vm_thread cycles could be an important factor to identify the performance bottlenecks.

In this study, all statistics of hardware event are divided into different groups by the types of JVM threads. Thus the correlation between JVM threads and bottlenecks of performance and scalability could be identified.

### 3.2 The Throughput Analysis

In order to observe the interactive performance scales with various numbers of cores and threads, the scales of throughput speedup could be used to examine the scalability of each benchmark. The throughput scaling of DaCapo benchmarks is shown in Fig. 1a. The throughput of each benchmark has a normalization value of 1.0 in a single core configuration.

In Fig. 1a, the improvement of throughput of Eclipse, Hsqldb and Xalan could be observed with the use of more cores. It is worth noting the poor scalability of Hsqldb. It seems that the lock contention problem which related to the behavior of a database application limits the advantage of using more cores. Thus the scalability of Hsqldb is lower than Eclipse and Xalan. The limitation of scalability will be verified in Section 3.3.2.

On the other hand, no any advantage of multicores is observed in Lusearch. The potential performance and scalability bottlenecks could offset the benefit of multicore systems. It seems that the poor cache utilization leads to the results, and it would be verified in Section 3.3.3.

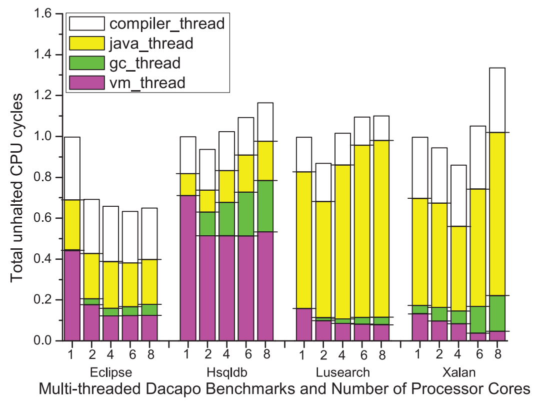


Fig. 2. Total cycles of DaCapo multithreaded benchmarks divided by JVM thread types.

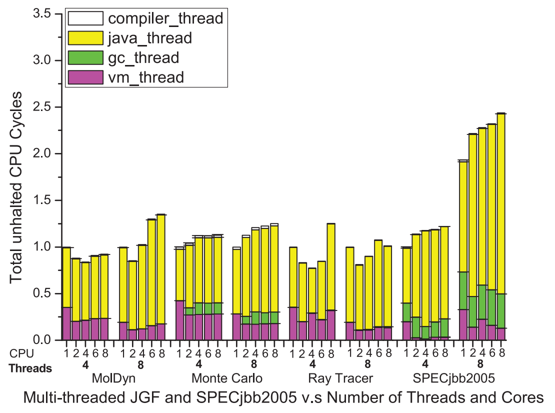


Fig. 3. Total cycles of the SPECJbb2005 and JGF multithreaded benchmarks divided by JVM thread types.

Moreover, there are not significant improvements of Eclipse and Hsqldb, when the number of cores is greater than four. However, Xalan has a good throughput improvement after six cores. We hypothesize that the good cache utilization might lead to good scalability in Xalan, and this hypothesis will be verified in Section 3.3.3.

Due to the maximum number of cores is eight in our study, the use of one to eight threads could present the application behaviors for our analysis. As shown in Figs. 1b, 1c, 1d, and 1e, the use of 16 and 32 threads has slight differences with the use of eight threads in JGF and SPECjbb2005.

The SPECjbb2005 throughput scaling with various numbers of cores and threads is shown in Fig. 1b. The throughput of single thread and a single core has normalized to 1.0, then, for each different configuration, the throughput is normalized to the configuration of single thread and a single core. In Fig. 1b, the performance improvements could be observed with the use of more cores and threads. That shows SPECjbb2005 can have the advantages from multithreading and multicore systems.

The throughput scaling of JGF multithreaded benchmarks are shown in Figs. 1c, 1d, and 1e. The performance peak could be observed when the number of threads equals to the number of cores. It shows the competition among the application threads for resource to execute is not intense. On the other hand, when the number of cores is greater than the number of threads, the possibility of a thread being assigned to a different core after a context switch is much higher. The core reassignment tends to generate the cache misses and the cache-to-cache transfer, and then degrade the cache utilizations.

The summary of throughput is as follows: First, Lusearch can not have advantage with the use of two and more cores. Second, the use of more cores and threads could improve the throughput in SPECjbb2005 and JGF. Finally, the performance peak could be observed when the number of threads equals to the number of cores in JGF multithreaded benchmarks.

## 3.3 Distributions of CPU Cycles

The analyses by breaking down the total unhalted CPU cycles are shown in this section. The total unhalted CPU cycles are the summary of all used cores in this study. Due to a single core is used for comparisons, the indicated cycle numbers are the averages of two, four, six, and eight cores in this section.

### *3.3.1 The Breakdown of CPU Cycles by JVM Threads*

The *total unhalted CPU cycles*, also known as *total cycles*, could be accounted by an architectural hardware event of the performance counter. Total cycles could be used to represent the consummation of CPU resource; thus it is an important factor for performance analysis.

The total cycles breakdown of DaCapo benchmarks is shown in Fig. 2. The single-core configuration has a normalization value of 1.0, then, for each benchmark, the total cycles are normalized to a single core.

First, total cycles increasing could be observed with the use of more cores in DaCapo benchmarks, except for Eclipse. It seems that there is a strong correlation between the decreases of total cycles and good scalabilities in Eclipse. This correlation will be further investigated via the cycle breakdown in Section 3.3.2.

Second, the vm_thread contributes mostly in Hsqldb. Moreover, the percentages of vm_thread cycles (51 percent) are higher than the other benchmarks (4-15 percent). That shows the high requirements of safepoints in Hsqldb. More use of safepoints could lead to more lock contentions, and reduce throughput.

Due to the largest live heap volume (72 MB) and largest number of live objects (3,223,276) [6], there are four full GCs occurred in one run of Hsqldb. The full GCs occupy 58 percent of total executing time in Hsqldb. That leads to a frequent use of safepoints; thus significant cycles could be observed by a vm_thread in Hsqldb.

Finally, the significant contributions of java_threads could be observed in DaCapo benchmarks (18 percent in Eclipse and Hsqldb, 53 percent in Lusearch and Xalan). The cycles of java_thread increase significantly with the use of more cores. It seems that java_threads contend with other java_threads, and then reduce cache utilizations. The bad cache utilization could be a bottleneck of performance, and it will be further investigated in Section 3.3.2.

The total cycles breakdown of SPECjbb2005 and JGF benchmarks is shown in Fig. 3. Due to the large number of data which comes from various configurations of threads and cores, the statistics of four and eight threads are presented in following sections. The configuration of four threads and a single core has a normalization value of 1.0, then, for other configurations, the total cycles are normalized to four threads and a single core.

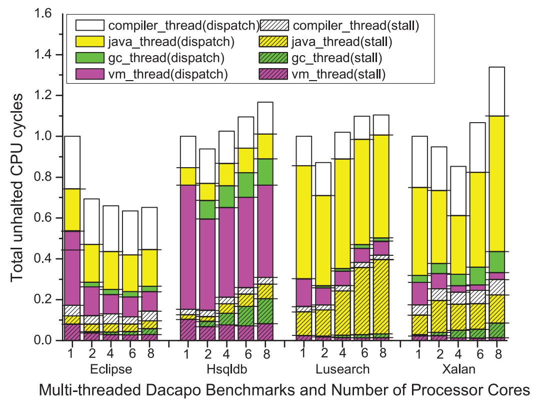


Fig. 4. Total cycles of the DaCapo multithreaded benchmarks divided by stall and dispatch cycles and JVM threads.

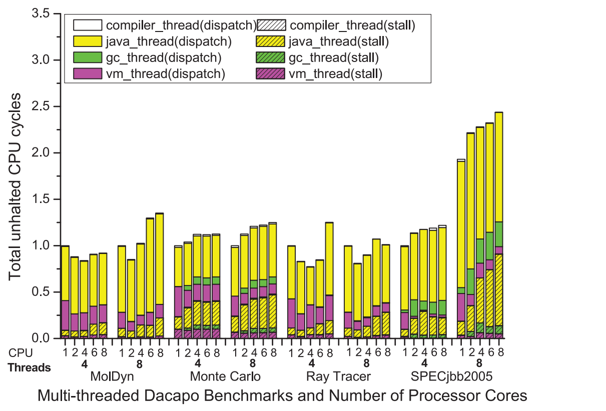


Fig. 5. Total cycles of SPECjbb2005 and JGF benchmarks divided into stall and dispatch cycles and JVM threads.

In Fig. 3, the vm_thread cycles of eight threads (15 percent) are lower than four threads (31 percent) in JGF benchmarks. That shows fewer requirements of safepoints and fewer lock contentions with the use of more threads. Thus the configurations of eight threads could have better scalability than four threads in JGF benchmarks.

Based on the analyses, it seems that a vm_thread plays an important role in the issue of scalability. Moreover, among all benchmarks except Hsqldb, java_threads contribute total cycles mostly. Thus the further analyses of vm_thread and java_thread are performed to observe the potential bottlenecks in the next section.

### 3.3.2 The Breakdown of CPU Cycles by Stall Cycles

Total cycles could be divided into two groups, dispatch and stall. After instructions are decoded into the executable micro operations, the *Reservation Station* (RS) issues the appropriate micro operations to *Execution Units*. The micro operations would be dispatched when resources are adequate. The CPU cycle of dispatched operations is dispatch cycle. On the other hand, when micro operations stall, the penalty cycle is stall cycle.

Based on this correlation, the summary of stall cycles and dispatch cycles equals to total cycles, the total cycles could be breakdown for more information. The total cycles breakdown of DaCapo benchmarks is shown in Fig. 4, and observations are listed as follows:

In Hsqldb and Lusearch, the stall ratio (stall cycles/ dispatch cycles) of java_threads increases with the use of more cores. The large stall ratio of java_thread leads to the lower utilization of the CPU resource and could be the bottleneck of Hsqldb and Lusearch. However, the reasons which lead to the high stall ratio are different. In Hsqldb, the feature of database access leads to the high lock contention. Thus, the stall ratio becomes high because each thread waits for another.

The large values of dispatch and stall java_thread cycles could be observed in Lusearch in Figs. 2 and 4. It seems that the most of CPU cycles are spent on the workload of Lusearch. Furthermore, the large values of java_thead stall cycles indicate the workload cannot be done well. Thus we may hypothesize that application threads of Lusearch cannot finish the certainly job due to some reasons, such as lock contentions. This hypothesis will be verified in Section 3.5.

Due to the shortest live time of Xalan's objects in DaCapo [6], the objects of Xalan die quickly. Thus the objects which die young, do not need to be transferred from caches to caches, and then the reductions of cache-to-cache transfers could be observed. With the use of more cores, more application threads work with the same amount of objects at the same time. Thus the lower cache-to-cache transfers could be observed in six and eight cores.

The total cycles breakdown of SPECjbb2005 and JGF benchmarks by stall and dispatch cycles is shown in Fig. 5. The variations of stall cycles, which are between four and eight threads, are insignificant in JGF (lower than 2 percent). On the other hand, the stall cycles of eight threads (41 percent) are almost twice as large as stall cycles of four threads (18 percent) in SPECjbb2005. We hypothesize that the lock contention of eight threads leads the high stall cycles in SPECjbb2005, and it will be verified in Section 3.5.

Based on the study of dispatch/stall cycles breakdown, the stable dispatch cycles could be observed after the use of two cores. That shows the workload of each run is similar. However, the increases of stall cycles usually could be observed with the use of more cores, and lead to the reductions of throughputs. This observation shows that the numbers of total cycles are not related to throughputs directly, but the variations of stall cycles could be an important factor to affect throughputs.

In order to detail the hardware events which lead the stall cycles, the breakdown of stall cycles by hardware events would be detailed in next section.

### 3.3.3 The Breakdown of Stall Cycles

Since the stall cycles could be an import factor to affect the performance and scalability. Thus the analysis of hardware stall events which lead to stall cycles could be used to identify potential bottlenecks. Based on the research of cycle accounting analysis [10], the most of stall events are related to the memory latency. Therefore, the hardware stall events, which are due to the memory latency, would be analyzed in this section.

The first stall event is the *L2 cache miss*. It means the CPU cycles stall is due to an L2 cache miss. The largest penalty cycles leads to the L2 miss contributes mostly of stall cycles. The second one is the *cache-to-cache transfer*. A cache-to-cache transfer means the shared data, either the success

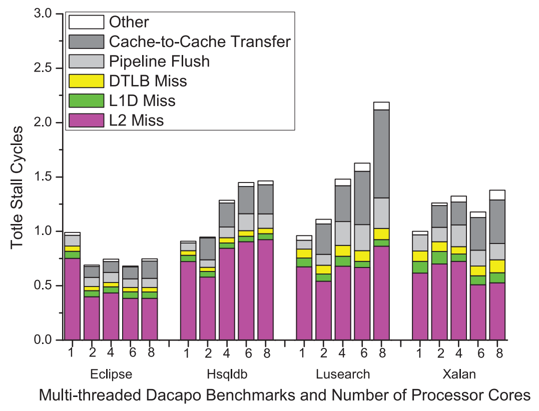


Fig. 6. The total stall cycles of DaCapo multithreaded benchmarks divided by hardware stall events.

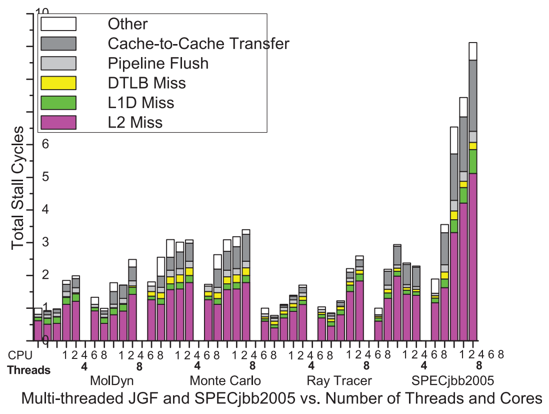


Fig. 7. The total stall cycles of the JGF and SPECjbb2005 multithreaded benchmarks divided by stall events.

sharing or false sharing, has to be copied to the main memory or to the other cache before it can be used. In addition, other stall events are included in our statistics, such as *L1 data cache miss*, *Pipeline flush*, and *DTLB Miss*. The stall cycles of above events are represented as $SC_{L2}$, $SC_{c2c}$, $SC_{L1}$, $SC_{pf}$, and $SC_{dtlb}$ in this study.

The stall cycles of DaCapo benchmarks breakdown by stall events are shown in Fig. 6. The $\mathrm{SC_{L2}}$ contributes mostly of total stall cycles (77 percent in Eclipse, 69 percent in Hsqldb, 58 percent in Lusearch and 53 percent in Xalan). It seems that there is a strong correlation between the low stall cycles of L2 cache misses and good scalabilities. Moreover, the significant reduction of $\mathrm{SC_{L2}}$ after four cores could be observed in Xalan. The reduction leads to the improvement of six and eight cores in Xalan. Therefore, the reduction of $\mathrm{SC_{L2}}$ could be an important key to improve scalabilities.

It is worth noting that the $\mathrm{SC_{c2c}}$ increases with the use of more cores in Lusearch significantly. That shows a large number of true and false sharing of data among different cores. Moreover, the lock contention which is between threads might aggravate it, and then cancels out the benefit of the use of more cores.

The stall cycles breakdown of SPECjbb2005 and JGF benchmarks is shown in Fig. 7. Due to a large volume of objects is allocated during executions, the large cycles numbers of $\mathrm{SC_{L2}}$ and $\mathrm{SC_{c2c}}$ could be observed in MonteCarlo (31 percent) and SPECjbb2005 (342 percent). On the other hand, the small memory footprint leads to the few cycle numbers of $\mathrm{SC_{L2}}$ and $\mathrm{SC_{c2c}}$ in MolDyn (22 percent) and RayTracer (11 percent). That leads to the better scalability of these benchmarks.

With the use of more cores or threads, the L2 cache performance could be worse. Two factors might be related with this issue. First, the default memory allocator of HotSpot allocates objects in nursery heap space through a bump pointer. The objects belonging to different threads could be allocated close to each other. The access to one object might lead to the other object to be loaded into the same cache line. Obviously, this will worsen the spatial cache locality for the current thread.

Second, the memory allocator could lead to high cache-to-cache transfers. If two or more cores try to access data in the same cache line but in different processor caches simultaneously, writing data in one cache might lead to the data to be invalidated in other processor caches. Therefore, the use of more cores and more application threads could lead to this situation even worse due to the increasing contentions of memory accesses.

The further investigation shows that using a thread-local heap or allocation buffer might improve cache system performance on multicores systems potentially. In order to observe this potential, the analyses of memory components performance are performed in Section 3.4.

## 3.4 Memory System Performance

In order to detail the causes of bottlenecks, the memory system performance would be studied in terms of JVM thread types. Based on the analyses of stall cycles, two important factors, L2 cache misses and cache-to-cache transfers, would be studied in this section.

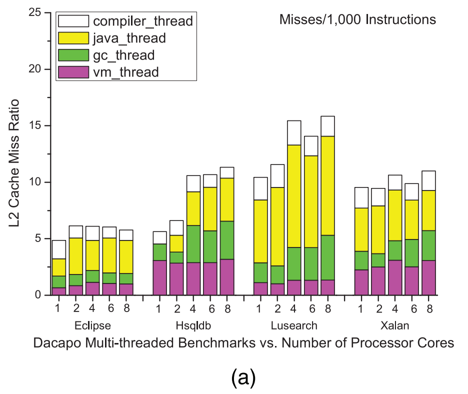
### 3.4.1 L2 Cache Misses

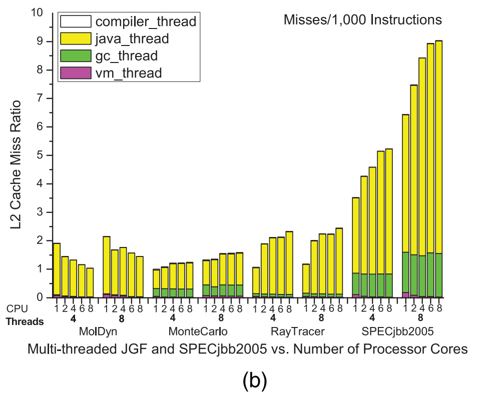
The L2 cache misses ratios are break down by JVM threads in Figs. 8a and 8b. It seems that not only java_thread contributes mostly of $SC_{L2}$, but also gc_thread does. Moreover, the benchmarks which have the large volume of objects usually lead to high $SC_{L2}$ of gc_threads. That shows the possibility of garbage collector improvements. We hypothesize that the default concurrent collector might not take advantage of multicore systems and multithreaded applications. This hypothesis will be verified in Section 3.6, and JVM tuning techniques are proposed in Section 3.7.

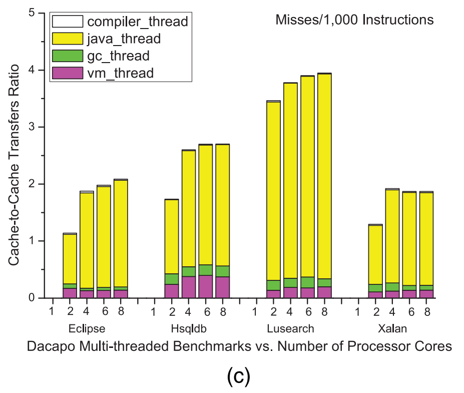
The increases of L2 cache misses with the use of more cores are observed in DaCapo benchmarks, especially in Hsqldb and Lusearch. Due to the database access behavior in Hsqldb, the data is reallocated frequently between each core and waited to be processed. The reallocation leads to an increase of L2 cache misses. On the other hand, the large numbers of L2 cache misses are observed in Lusearch. The bad cache utilization of Hsqldb and Lusearch could be considered as the lock contention problem, and it will be detailed in Section 3.5.

### 3.4.2 Cache-to-Cache Transfer

To simplify this analysis, cache-to-cache transfers are correlated to the thread responsible for data modifications. In Figs. 8c and 8d, the cache-to-cache transfer ratio would be analyzed by contributions of JVM thread types







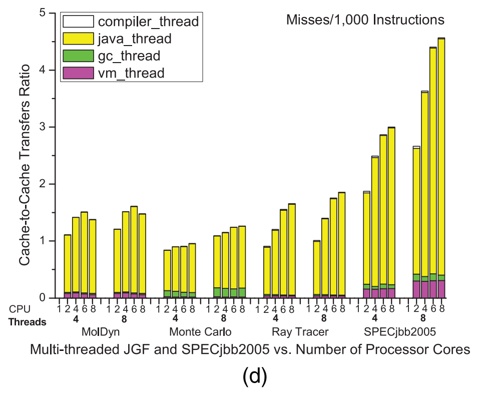


Fig. 8. The memory system performance. (a) The L2 cache miss ratio of DaCapo benchmarks. (b) The L2 cache miss ratio of SPECjbb2005 and JGF. (c) The cache-to-cache transfer of DaCapo benchmarks. (d) The cache-to-cache transfer of SPECjbb2005 and JGF.

First, it seems that the most of cache-to-cache transfers are dominated by java_threads. In addition, the use of more cores or threads often leads to an increase of the cache-to-cache transfer ratio. These observations show that cache-to-cache transfers share the similar behavior as L2 cache misses.

However, the causes which lead to L2 cache misses and cache-to-cache transfers are different. L2 cache misses are due to poor cache localities which are resulted from interleaved object allocations among threads mostly. On the other hand, true/false sharing of data among each core leads to cache-to-cache transfers mostly. It is worth noting that false sharing can be avoided, but not true sharing. Some related works study on this issue [15]. In order to narrow down the research scope, the total effect of cache-to-cache transfers is analyzed in this study.

Second, the low numbers of cache-to-cache transfers in Xalan could be observed in Fig. 8c. The short life span of Xalan's objects [6] leads to less cache-to-cache transfers since these objects tend to die young.

Finally, it seems that Lusearch has the large numbers of cache-to-cache transfers than other DaCapo benchmarks. This observation verifies the hypothesis in Section 3.3.3, the large stall cycles are resulted from the cache-to-cache transfer in Lusearch. Thus the cache-to-cache transfer could be considered as a bottleneck of Lusearch.

It is worthy noting that cache-to-cache transfers could be reduced significantly by scheduling threads into the previously running core. The parameter, *cpus_allowed*, is modified in data structures of a Linux thread to implement this approach, and experimental results are shown in Fig. 9. The significant reductions of cache-to-cache transfers could be observed in Lusearch (42 percent). Moreover, the 28 percent throughput improvements could be observed by this approach.

## 3.5 Lock Contention

In Java, the lock contention arises when multiple threads try to access a shared object exclusively at the same time. In general, synchronizations are used to guarantee a shared object being accessed correctly. There are two typical types of synchronizations, blocking and spinning. These two types of synchronizations have been applied to various synchronization mechanisms, such as mutual exclusion locks, condition variables, semaphores, spin locks, etc.

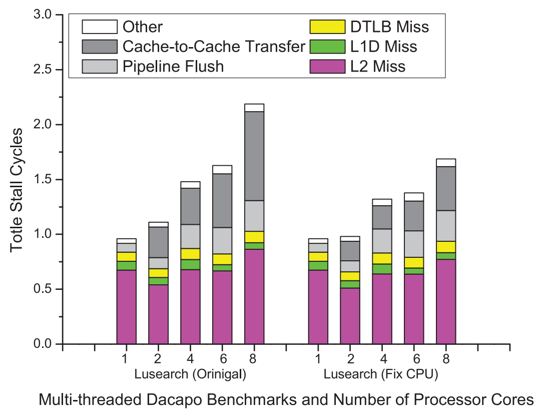


Fig. 9. Total stall cycles of Lusearch.

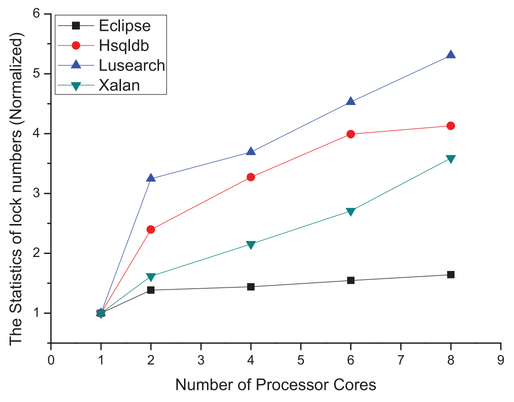


Fig. 10. The statistics of lock numbers of DaCapo multithreaded benchmarks versus numbers of processor cores.

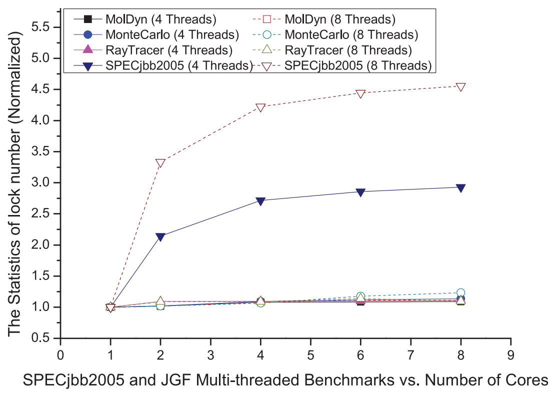


Fig. 11. The statistics of lock numbers of SPECjbb2005 and JGF multithreaded benchmarks versus numbers of processor cores.

The synchronization mechanism in Java monitor, is widely used in various JVMs. It will be inflated to a fat-lock when threads try to get exclusive accesses to shared objects simultaneously. In Java, a fat-lock could be considered as a pair of monitors. Instead of a fat-lock, a thin-lock could be considered as an implement of CAS (*Compare and Swap*). The proposition of a thin-lock is based on the observation that the locks are rarely contended in Java applications. When the competitions between threads are not contended, the use of a thin-lock is faster than the use of a fat-lock.

When objects are required by threads simultaneously, threads could be blocked and led to lock contentions. The lock contention is a complicated problem [16]. The statistics of lock numbers, including static locks and dynamic locks, are used to analyze this problem. The lock numbers of each benchmark has a normalization value of 1.0 in a single core configuration.

In Fig. 10, the significant increases of lock numbers could be observed in Lusearch with the use of more cores. Due to the text searching characteristic of Lusearch, multiple application threads might share a part of the text objects with the use of locks. This characteristic leads to threads wait for others to obtain locks and do searching. Thus the lock contention of Lusearch can be observed in Fig. 10. In addition, the problem of lock contention leads to a long waiting time for other threads. Thus performance degradations could be observed with the number of cores scales up in Lusearch.

The lock numbers of SPECjbb2005 and JGF are shown in Fig. 11. The slight increases of lock numbers are observed with the use of more cores or threads in JGF. On the other hand, the lock numbers of SPECjbb2005 are much higher than JGF benchmarks. Due to the large volumes of objects in SPECjbb2005, the overhead of minor garbage collections might be the cause which leads to large numbers of locks. The overhead could be a bottleneck of scalability in SPECjbb2005, and it will be verified in Section 3.6.

The investigation shows that Eclipse, Xalan, and JGF benchmarks could be classified as benchmarks with slight lock contentions. In these benchmarks, most accesses to shared objects could be done through thin locks. In very infrequent cases that fat locks have to be acquired, the average time of acquiring and holding a lock is shorter than other benchmarks. As a result, the low values of numbers of locks could be observed even if a large number of cores or threads are used.

On the other hand, Hsqldb, Lusearch, and SPECjbb2005 could be classified as benchmarks with heavy lock contentions. The tournament-based barriers contribute to this effect. Threads have to wait for particular barriers until all threads reach it. In order to relief the lock contentions, the workload and time slice must be well balanced among the application threads. Otherwise, a large number of CPU cycles would be wasted for waiting (actually spinning because yielding is used). However, in a heavy threaded environment, the well balancing cannot be guaranteed.

In conclusion, the lock contentions could degrade the performance and limit the scalability. Particularly, a large synchronization scope of a lock usually shows a strong limitation on scalability. In order to improve the scalability of multithreaded applications, the scopes of synchronizations should be minimized. Thus the time of acquiring or holding a lock could be reduced, and then better scalabilities could be reached through higher thread-level parallelism.

## 3.6 The Overhead of Minor Collections

The certain memory space is located as the heap when a Java application initiates. This memory space is a repository for live objects, dead objects, and free memory. The heap could be divided into three parts. They are young, old, and permanent generations. The young generation is optimized for objects that have a short lifetime relative to the interval between garbage collections. The objects, which are survived after several garbage collections, would be moved from the young generation to the old generation by *minor collections*.

When a multithreaded Java application executes with a multiple cores system, multiple threads create objects at the young generation simultaneously. Thus, the young generation could be filled up quickly. Moreover, the lock contention could lead to that threads have to wait for other threads. It could be the reason which leads to large numbers of locks in SPECjbb2005 and Lusearch. It is worth noting that objects could not be collected while the other living threads still hold them. The contention reduces the efficiency of garbage collection and leads to the significant overhead of minor collections.

In order to identify the overhead of minor collections, SPECjbb2005 is used to examine the variations of overhead.

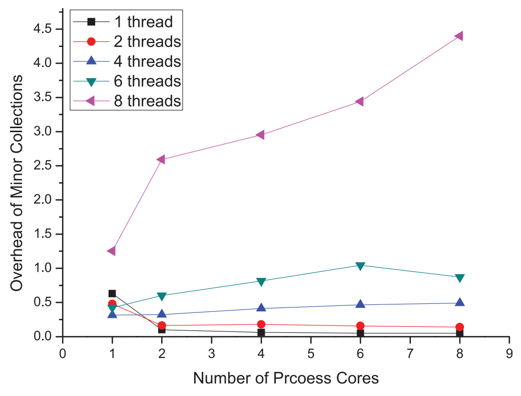


Fig. 12. The minor collection overhead of SPECjbb2005.

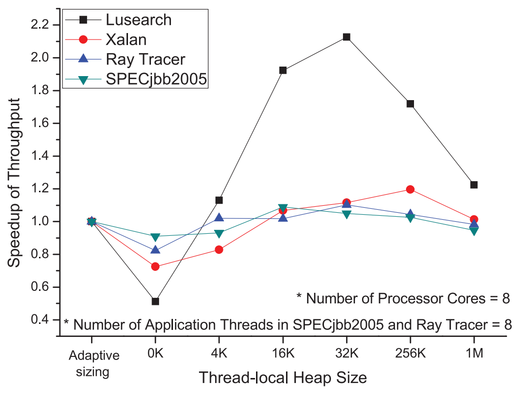


Fig. 13. Throughput versus Size of thread-local allocation buffer.

In Fig. 12, the significant overhead of eight threads could be observed. The significant overhead of eight threads might cancel the benefit of the use of more cores and threads, and could be a bottleneck of scalability.

The increase of the heap size could reduce the overhead directly. When the size of heap increases, the young generation also increases to hold more objects. However, the heap size cannot increase without a limit. Base on the premise of a same heap size, finding an appropriate size of young and old generation could be an important tuning technique, and this issue would be examined in Section 3.7.3.

## 3.7 JVM Tuning Techniques

In this section, three JVM tuning techniques are proposed to improve the scalability and performance for the observed bottlenecks. These tuning techniques would be examined to observe their improvements individually.

### *3.7.1 The Use of Thread Local Heap/Allocation Buffer*

Based on the studies in Section 3.3.3, memory stalls could be the important reason of performance degradations and scalability limitations. It seems that the high-level software behaviors could lead to the stresses on memory systems. For example, the multiple java_threads allocate and access objects simultaneously. The gc_threads scan objects and copy live ones from a nursery space to a mature space continuously. Thus the use of a technique, which improves the performance of memory systems in software levels, could be very important.

*Thread-local heap* (TLH) is a memory management scheme. With the use of TLH, each thread receives a partition of a heap for thread-local object allocations and thread-local garbage collections without synchronizations with other threads. The intention of TLH approach is to reduce the heap contention for object allocations and garbage collections [2]. However, the use of TLH scheme could improve the memory performance significantly with the use of multicores systems. Thus the study of thread local heap could be important to improve the performance of memory systems.

HotSpot JVM offers a similar approach, called *thread-local allocation buffer* (TLAB). There are some differences between a TLAB and a TLH. The TLH approach maintains the invariant that objects within each TLH are local to a single thread. All objects may be allocated into the TLH, as long as they are evacuated before becoming nonlocal. It has the advantage that thread-local objects could be collected independently without stopping other threads. However, TLH is complicated because it requires the support of compilers, and its overhead such as a write barrier have to be introduced.

In contrast, the TLAB approach allows any object to be allocated locally in the allocation buffer belonging to the thread which creates the objects. Moreover, the implementation of TLAB approach is simpler and often leads to the similar performance as a TLH. Thus, we focus on the TLAB instead of the TLH in this section.

The TLABs setting of Hotspot can be determined by three JVM options, *UseTLAB*, *ResizeTLAB*, and *TLABSize*. In this experiment, the adaptive TLAB allocations could be enabled by ResizeTLAB option. The user-defined TLAB sizes could be adjusted by TLABSize option. Furthermore, the use of a TLAB could be disabled when the UseTLAB option is set to false.

In order to evaluate the performance variations of the use of TLAB, various sizes of a TLAB are applied with four benchmarks to observe the throughput variations. The throughput scaling is normalized to 1.0 of the execution with adaptive TLAB sizing, default garbage collector, and the use of eight cores. The experiment results are in Fig. 13.

Based on the observations in Fig. 13, it seems that either large or small TLAB sizes cannot lead to significant improvements of performance. With the use of small TLAB sizes, a thread has to request a new buffer frequently when the current allocation buffer is full. The frequent memory allocations could lead to the contention on object allocation and memory fragmentation. Moreover, a small size of TLAB also leads the objects to be allocated into separate allocation buffers, and leads to the poor spatial locality. On the other hand, with a large size of TLAB, the performance could be offset by the penalties such as TLB misses. Based on experimental results, the TLAB sizes which are between 16 KB and 256 KB might lead to significant performance improvements.

In addition, the 22 percents of performance improvements could be observed averagely with the use of adaptive TLAB sizing in Fig. 13. However, comparing with experimental results of the use of other TLAB sizes, the adaptive TLAB sizing could be enhanced in the future for particular applications.

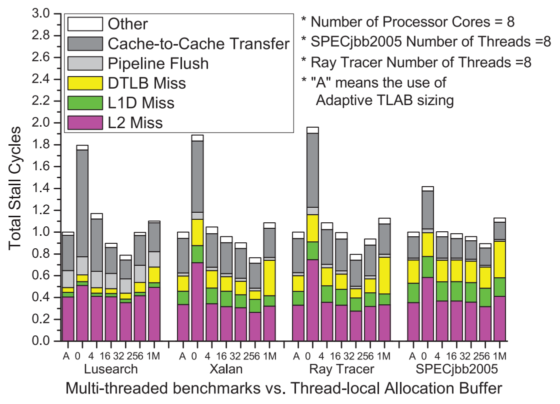


Fig. 14. Stall cycles breakdown versus thread-local allocation buffer.

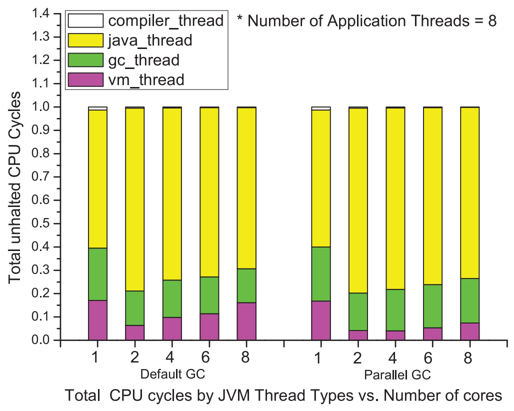


Fig. 15. Total cycles of SPECjbb2005 versus parallel and default GC.

In order to detail the improvements of cache performances, the bearkdown of stall cycles is shown in Fig. 14 by various stall events. As shown in Fig. 14, the considerable reductions of stall cycles could be observed with the use of various TLAB sizes. The fewer stall cycles indicates better CPU resource utilizations, and then leads to better throughputs.

Objects belong to different threads could be placed in the same cache line but in different core. Thus writing to this cache line in one core could cause the data to be invalidated in caches of other cores. Therefore, the significant increases of cache-to-cache transfers and L2 cache misses could be observed with the use of more cores. However, the use of TLAB alleviates this problem by allocating thread-local objects together. Thus cache locality could be enhanced significantly due to most cache lines would not be shared among threads. The enhanced cache locality could be observed by the reductions of cache-to-cache transfers and L2 cache misses in Fig. 14.

The use of appropriate size of a TLAB could enhance the performance of local objects accessing. Thus the lock contention could be reduced by the better hit ratio of local objects. It is worth noting that the appropriate size of a TLAB could depend on the cache organization, the application behavior, etc. Furthermore, the use of adaptive TLAB sizing does not reach the lowest stall cycles among the other TLAB configurations. This observation confirms the observations in Fig. 13. Thus dynamically choosing the size of a TLAB has the potential to further performance improvements.

### 3.7.2 The Use of Parallel Garbage Collector

In order to decrease the overhead of garbage collections and increase the throughputs, The *Parallel Garbage Collector* (PGC) is proposed in the HotSpot JVM. The operations of a parallel collector could be executed by all available cores on multicore systems. Thus the better performance of garbage collections could be reached.

The SPECjbb2005 benchmark is used to examine the effect of a PGC due to the large volume of objects. The scaling of total cycles is proposed in Fig. 15 with the use of various numbers of cores. In order to compare the proportions of each JVM thread, the numbers of cycles are normalized to 1.0

Due to the contention and synchronization issues among gc_threads, the PGC consumes more CPU cycles than the CGC slightly. However, the reductions of vm_thread cycles could be observed with the use of a PGC in Fig. 15. As we mentioned in Section 3.3.1, the increase of vm_thread cycles usually indicates the degradation of performance due to the safepoint using in a vm_thread. Therefore, the reduction of vm_thread cycles s could be considered as the reductions of idle cycles, and then leads to performance improvements with the use of a PGC.

### 3.7.3 The Use of the Ratio of New to Old Generation

The significant overhead of minor collections is observed in Section 3.6, and it could be a performance bottleneck for multithreaded applications. In order to improve this bottleneck, the JVM tuning technique, the adjustable ratio of the new to old generation, to reduce the overhead.

Based on the studies in Section 3.6, the objects which are preserved in the young generation cannot be collected due to the threads still hold them. The young generation could be filled with the living objects quickly. In order to keep the living objects and spare space for more new objects, the living objects must be swept from the young generation to the old generation by minor collections frequently. Frequent minor collections could lead to heavy overhead, and then reduce the throughput.

In order to reduce the overhead, the size of young generation and old generation should be configured well. Comparing with the same size of a heap, a larger nursery space could keep more living objects. Due to a larger nursery defers collections, and then provides objects more time to die. In addition, the overhead of minor collections is due to copying objects in the heap, thus the use of TLAB does not reduce the overhead of minor collections significantly.

The JVM tuning technique, the *NewRatio*, could be used to configure the ratio of new to old generations in a HotSpot JVM. The experiment is proposed to evaluate the performance of the NewRatio tuning technique. The SPECjbb2005 is selected for this experiment due to that the bottleneck of SPECjbb2005 is the overhead of minor collections. The default value of NewRatio in the server mode is two. Thus the values of NewRatio are examined from 2 to 26 in this experiment.

The throughput scaling of SPECjbb2005 is shown in Fig. 16. The results are normalized to 1.0 with the use of a single thread and the default value of NewRatio. The

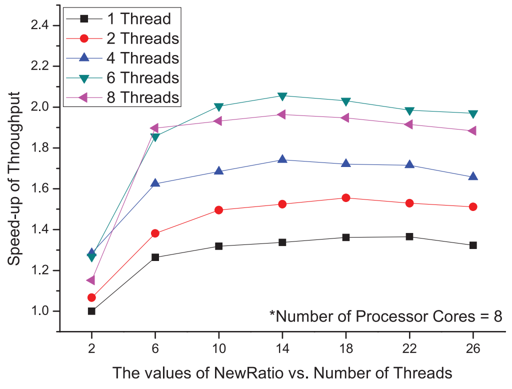


Fig. 16. The throughput of SPECjbb2005 versus various NewRatio.

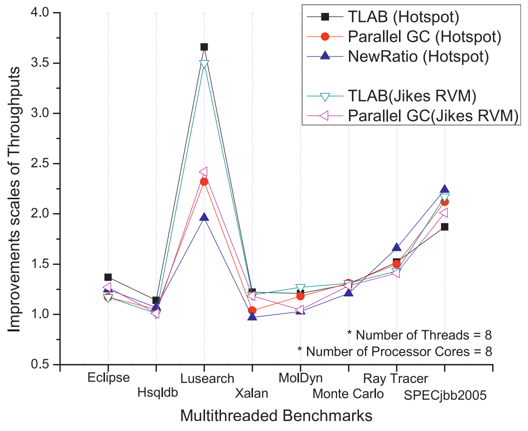


Fig. 17. The improvements of JVM tuning techniques.

average 45 percents improvements of throughputs could be observed when the NewRatio values are between 10 and 18. This observation shows the appropriate NewRatio values could improve the throughput significantly. It is worth noting that the throughput of six and eight threads is better than four threads. That shows the appropriate NewRatio values could improve the bad scalabilities of the use of six and eight threads in SPECjbb2005.

### 3.7.4 Validation of JVM Tuning Techniques

In order to verify our findings in this paper, the validations of JVM tuning techniques are proposed in this section. Three JVM tuning techniques are applied on eight multithreaded benchmarks to evaluate the throughputs in this experiment. The throughput scaling of benchmarks are shown in Fig. 17. The throughput scaling is normalized to 1.0 of the execution with default JVM option, default garbage collector and eight cores for DaCapo multithreaded benchmarks. In addition, the number of cores is set to eight to examine the scalability of different thread numbers in SPECjbb2005 and JGF multithreaded benchmarks.

First, the performance improvements of the use of TLAB could be observed among the most of benchmarks, such as Eclipse, Lusearch, Xalan, MolDyn, and MonteCarlo. It is worth noting that the four times improvements of performance could be observed with the use of TLAB in Lusearch. On the other hand, due to the lock contentions could degrade the performance, only slight throughput improvements could be observed in Hsqldb. The results verify the findings in Sections 3.4.2, 3.5, and 3.7.1.

Second, the performance improvements of the use of a PGC could be observed with the high threaded benchmarks which allocate large volume of objects, such as MonteCarlo, RayTracer, and SPECjbb2005. In general, the use of PGC does not lead significant improvements with the use of fewer cores or the slight requirement of garbage collections.

Third, the significant performance improvements could be observed as the NewRatio tuning technique is used in RayTracer and SPECjbb2005. In SPECjbb2005, the use of an appropriate value of NewRatio leads to the twice improvements of throughputs. That shows the use of NewRatio could reduce the overhead of minor collections efficiently. With the increase of object volumes, the improvements could become significant. This observation verifies the findings about the overhead of minor collections in Section 3.7.3.

In order to perform the comparison of different JVMs, the JVM tuning techniques are applied on Jikes RVM to evaluate the improvements. Due to the NewRatio option is not available in Jikes RVM, for the repeatability of experiments, the NewRatio technique is not used on Jikes RVM in this study.

As the observation of Jikes RVM in Fig. 17, the improvements of each tuning technique on Hotspot and Jikes RVM are similar. The differences of improvements by each optimization technique are less than five percent on Hotspot and Jikes RVM. These results suggest that the proposed tuning techniques are applicable to various JVMs.

Moreover, in order to clarify the scalability issue, the variations of scalability with the use of JVM tuning techniques are shown in Fig. 18. The single core or single thread configurations have a normalization value of 1.0, then, for each benchmark, the throughputs are normalized to a single core or a single thread.

In Fig. 18, the significant improvements of scalability could be observed in Lusearch, Xalan, and SPECjbb2005. The use of TLAB improves the scalability near linear in Lusearch and Xalan. That shows the appropriate TLAB sizing could reduce the lock contentions significantly, and then leads to the well utilizations of the use of multiple cores.

In SPECjbb2005, the scalability which is near linear could be observed with the use of NewRatio options. That shows the overhead of minor collections, which is the reason leads to the poor scalability with the use of six and eight threads in SPECjbb2005, could be reduced significantly by the NewRatio tuning technique.

Overall, the validations show that JVM tuning techniques, including an appropriate TLAB, NewRatio and a parallel garbage collector could improve the performance and the scalability. The significant performance improves could be observed in Lusearch, Xalan, RayTracer, and SPECjbb2005. It shows these JVM tuning techniques are suitable for the highly threaded and memory-intensive Java applications. In terms of the ranges of improvements, the use of TLAB and NewRatio could lead to greater effects than the use of a parallel collector. The result also implies that the enhancement of cache performance would have the potential for performance and scalability improvements.

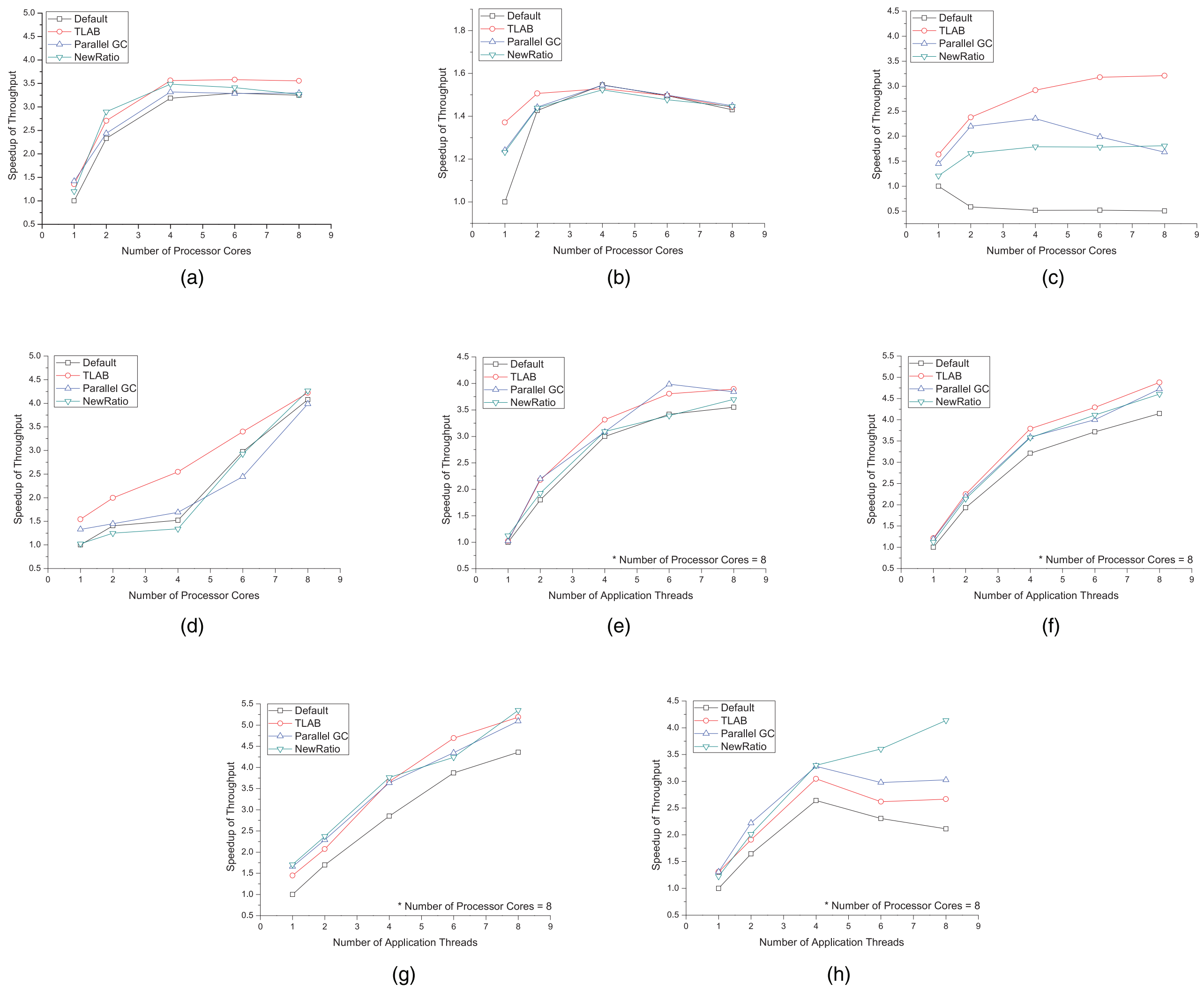


Fig. 18. The throughput scaling of Java multithreaded benchmarks versus optimization techniques. (a) Eclipse, (b) Hsqldb, (c) Lusearch, (d) Xalan, (e) MolDyn, (f) MonteCarlo, (g) RayTracer, (h) SPECjbb2005.

## 4 Related Work

The behaviors of Java applications have been evaluated since Java was first introduced in late 1995 [13]. Most studies focused on single threaded Java programs, especially the SPECjvm98 benchmarks. However, studies of multithreaded benchmarks are rare. In recent years, due to the popularity of Java-based server applications, the performance of Java multithreaded programs is becoming an issue of great interest.

The performance counters are used to evaluate the characterizations of Java server applications with the use of the uniprocessor system. Yue et al. studied the impacts of various numbers of Java threads on the microarchitecture [5]. Instead of running benchmarks on a uniprocessor system, our work focuses on the performance characterizations of multithreaded Java programs on multicore systems. Many studied in our work, such as cache-to-cache transfer, are not available on the uniprocessor systems.

Sweeney et al. [11] recently reported a performance monitoring system in Jikes RVM, which was implemented based on the hardware performance counters [13]. As a demonstration, two performance issues (including general performance trends and the memory latency issues) were investigated with the use of this system. The result shows that their tool is able to attribute the observed program behaviors to the specific components of a JVM. However, the profiling system has some limitations. The performance could be examined heavily rely on the capability, which is provided by the performance counters of processors.

Hauswirth et al. [12] introduce an infrastructure to further examine applications of their profiling systems. This research introduces a technique, vertical profiling, to correlate the performance characterizations across the layers of modern object-oriented systems. Different from our work, their research did not particularly focus the scalability issues on multicore systems.

It is worth noting that a biased lock is used to prevent executing expensive atomic calls by letting a lock being biased towards a particular thread [14]. When the thread acquires or releases the lock, no atomic operations are required. Only when another thread acquires the lock, an atomic call is made. Thus the use of biased locking could reduce the synchronization overheads.

Based on the studies in this paper, we observe that most objects are locked by at least one thread during their lifetime. Therefore, the sensible case could be optimized. A biased lock allows that a thread to bias an object toward itself. Once an object is biased, threads could lock and unlock this object subsequently without resorting to expensive atomic instructions. Thus the use of biased locking could reduce the synchronization overheads in Java significantly, and it is enabled by default in Hotspot JVM 1.7.

## 5 Conclusions

In this paper, the performance and scalability issues of multithreaded Java applications on multicores systems are studied. The detailed analyses by total cycles provide the information to evaluate the performance scaling of multithreaded benchmarks with the use of different number of cores and threads. The unique approach of analyses, correlating the low-level hardware performance event data to system software components, could be use to identify the performance and scalability bottlenecks at multiple levels. The summary of this paper is as follows:

First, the lock contentions could limit the performance and scalability potentially. The inappropriate use of synchronizations in a multithreaded Java application could lead to large numbers of stall cycles. Particularly, in a highly threaded environment, the heavy lock contentions usually lead to a strong limitation on the scalability.

Second, among memory access latency components, the most of memory stall cycles are produced by L2 cache misses and cache-to-cache transfers. Moreover, the use of more cores or threads, independently of each other, often leads to increases in L2 cache misses and cache-to-cache transfers. The low performance of memory systems could reduce the utilizations of multiple cores and threads applying.

Finally, in order to improve the problems of lock contentions and memory systems, the JVM tuning techniques are used to reduce the bottlenecks. With the validations of JVM tuning techniques, we observe that the use of an appropriate TLAB sizing could reduce L2 cache misses and cache-to-cache transfer significantly. The appropriate TLAB sizes, which are between 16 KB and 256 KB, usually lead to performance improvements. In addition, the dynamically choosing the TLAB size has the potential to further improve the performance. On the other hand, the use of the NewRatio, the ratio of new to old generations, could reduce the overhead of minor collections. The application of the NewRatio tuning technique could reduce the overhead and improve the throughputs and scalabilities significantly.

## Acknowledgments

This work was supported by the Taiwan National Science Council (NSC) under Grant No. NSC-096-2917-I-006-113. This work was conducted while Kuo-Yi Chen was a visiting student at Iowa State University. This material is partially based upon work supported by the US National Science Foundation (NSF) under Grant No. 0296131 (ITR), 0219870 (ITR), and 0098235. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the US National Science Foundation (NSF). The authors thank Dr. Palmer J. Holden, Emeritus Professor, Iowa State University, Ames, IA.

## References

[1] Lime Wire, LLC. Lime Wire, Web Document, http://www.limewire.org, 2007.

[2] T. Domani, G. Goldshtein, E.K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald, "Thread-Local Heaps for Java," *Proc. Third Int'l Symp. Memory Management (ISMM '02),* pp. 76-87, 2002.

[3] OpenJDK Project, https://openjdk.dev.java.net/, 2009.

[4] J. Donnell, "Java Performance Profiling Using the VTune Performance Analyzer," *Intel,* 2004.

[5] Y. Luo and L.K. John, "Workload Characterization of Multithreaded Java Servers," *Proc. 2001 IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS),* pp. 128-136, Nov. 2001.

[6] S.M. Blackburn, R. Garner, C. Hoffman, A.M. Khan, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," *Proc. 21st Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '06)* pp. 169-190, Oct. 2006.

[7] DaCapo Research Project, http://www.dacapo-group.org/, 2009.

[8] A. Adamson, D. Dagastine, and S. Sarne, "SPECjbb2005—A Year in the Life of a Benchmark," *Proc. SPEC Benchmark Workshop,* 2007.

[9] Intel VTune(TM) Performance Analyzer Documentations, http://software.intel.com/en-us/intel-vtune, 2009.

[10] *Cycle Accounting Analysis on Intel Core2 Processors.* Dr. David Levinthal PhD, Intel Corporation.

[11] P.F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind, "Using Hardware Performance Monitors to Understand the Behavior of Java Applications," *Proc. USENIX Third Virtual Machine Research and Technology Symp. (VM '04),* May 2004.

[12] M. Hauswirth, P.F. Sweeney, A. Diwan, and M. Hind, "Vertical Profiling: Understanding the Behavior of Object-Oriented Applications," *Proc. 18th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA),* Oct. 2004.

[13] R. Ramesh, N. Vijaykrishnan, L.K. John, and S. Anand, "Architectural Issues in Java Runtime Systems," *Proc. Sixth Int'l Symp. High-Performance Computer Architecture (HPCA),* Jan. 2000.

[14] S.M. Blackburn, P. Cheng, and K.S. McKinley, "Myths and Realities: The Performance Impact of Garbage Collection," *Proc. SIGMETRICS '04/Performance,* pp. 25-36, 2004.

[15] J. Torrellas, H.S. Lam, and J.L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," *IEEE Trans. Computers,* vol. 43, no 6, pp. 651-663, June 1994.

[16] R.L. Halpert, C.J.F. Pickett, and C. Verbrugge, "Component-Based Lock Allocation," *Proc. 16th Int'l Conf. Parallel Architecture and Compilation Techniques (PACT '07),* pp. 353-364, 2007.

**Kuo-Yi Chen** received the BE and MS degrees in engineering science from the National Cheng Kung University. He was a visiting student at the Iowa State University during 2007-2009. He is currently working toward the PhD degree in National Cheng Kung University. His research interests include Java virtual machines, multithreading, multicores systems, and Green computing. He is a student member of the IEEE.

**J. Morris Chang** is an associate professor at Iowa State University. Dr. Chang received his PhD degree in computer engineering from North Carolina State University. His industrial experience includes positions at Texas Instruments, Microelectronic Center of North Carolina and AT&T Bell Laboratories. He received the University Excellence in Teaching Award at Illinois Institute of Technology in 1999. Dr. Chang's research interests include: Wireless Networks, Performance study of Java Virtual Machines (JVM), and Computer Architecture. Currently, he is a handling editor of *Journal of Microprocessors and Microsystems* and the *Middleware & Wireless Networks* subject area editor of *IEEE IT Professional*. He is a senior member of IEEE.

**Ting-Wei Hou** received the BS, MS, and PhD degrees, all in electrical engineering from the National Cheng Kung University, Taiwan, in 1983, 1985, and 1990, respectively. Since 1990, he has been an associate professor in the Department of Engineering Science of the National Cheng Kung University. He was a visiting scholar of CSRD of the University of Illinois at Urbana-Champaign, Illinois, during 1993-1994. Since 2008, he has also been the director of the Department of Medical Informatics of the National Cheng Kung University Hospital. He was the project leader of the pilot project on Healthcare IC cards in Penghu, Taiwan from 1998 to 2003, which encouraged the National Healthcare IC card project. Since 1998, he has been researching on Java-based embedded systems. Currently, he is working on Java virtual machines, Java obfuscators, MHP, OSGi, and their applications in medical systems. His research interests include embedded systems and system integration. He is a member of the IEEE.