Allocation-Phase Aware Thread Scheduling Policies to Improve Garbage Collection Performance [*]

Feng Xian, Witawas Srisa-an, and Hong Jiang

University of Nebraska-Lincoln
Lincoln, NE 68588-0115
{fxian,witty,jiang}@cse.unl.edu

Abstract

Past studies have shown that objects are created and then die in phases. Thus, one way to sustain good garbage collection efficiency is to have a large enough heap to allow many allocation phases to complete and most of the objects to die before invoking garbage collection. However, such an operating environment is hard to maintain in large multithreaded applications because most typical time-sharing schedulers are not allocation-phase cognizant; i.e., they often schedule threads in a way that prevents them from completing their allocation phases quickly. Thus, when garbage collection is invoked, most allocation phases have yet to be completed, resulting in poor collection efficiency.

We introduce two new scheduling strategies, LARF (lower allocation rate first) and MQRR (memory-quantum round robin) designed to be allocation-phase aware by assigning higher execution priority to threads in computation-oriented phases. The simulation results show that the reductions of the garbage collection time in a generational collector can range from 0%-27% when compare to a round robin scheduler. The reductions of the overall execution time and the average thread turnaround time range from -0.1%-3% and -0.1%-13%, respectively.

Categories and Subject Descriptors D.3.4 [*Programming Language*]: Processors—Memory management (garbage collection)

General Terms Experimentation, Languages, Performance

Keywords Garbage collection, Thread scheduling

[*] This work was sponsored in part by the National Science Foundation through awards CNS-0411043 and CNS-0720757, and by the Army Research Office through DURIP award W911NF-04-1-0104.

1. Introduction

Over the past few years, we have seen widespread adoption of Java as a programming language for the development of long-running server and multithreaded applications [14, 16]. Java possesses many attractive features such as simple threading interfaces and type-safety, which ease the development of complex software systems. Moreover, it also uses *garbage collection* (GC) to simplify the task of dynamic memory management, which can greatly improve programmer productivity by reducing memory errors. However, a recent study has shown that garbage collection can be inefficient in long-running multithreaded server applications facing unexpected heavy demands. In these circumstances, the throughput performances of these applications can degrade sharply, resulting in failure with little or no warning [12, 32].

In our previous work [32], we observed that as the workload of a server application increases, so do the lifespans of objects in that application, leading to poor GC performance. Further investigation reveals that thread scheduling policies play a major role in causing the object lifespans to increase [31]. Currently, most modern operating systems employ some forms of preemptive round robin scheduling policies based on time-quantum. Such policies are designed to provide fairness in terms of execution time, but *do not consider allocation phases* as part of the scheduling decision.

An *allocation phase* is an execution segment, in which an application allocates a large volume of objects. A study by Wilson and Moher [30] reports that allocation phases often occur during interactive segments of the program, and garbage collection can be efficiently invoked at the end of non-interactive segments or computation-oriented phases [28]. Our studies have shown that when a server application is facing heavy workload, garbage collection is often invoked when threads are in these allocation phases, instead of the more ideal computation-oriented phases [30].

To understand why GC is often invoked when threads are in allocation phases, we investigate the events that take place when a server application is under stress. In server applications, more threads become active as demands increase. A larger number of threads also means that there is more competition for the CPU time. Thus, a thread often has to wait longer for its turn to execute. Because these threads share the same heap, object allocations can result in much higher needs for heap memory, leading to more garbage collection invocations. As a result, an execution interval between two GC invocations becomes shorter. (We refer to each execution interval as a *mutator* interval.)

In time-quantum based scheduling, the combination of interleaved executions, shorter mutator intervals, and longer wait times can cause threads to get fewer execution quanta in each mutator interval. Because an allocation phase often takes many quanta to complete,[1] these threads may not be able to complete many, if any, allocation phases prior to a GC invocation. The delay in completing these allocation phases makes the lifespans of these objects appear to be much longer.

This Work. We propose two new policies that consider allocation phases as a thread-scheduling criterion. The first policy is *Lower Allocation Rate First* or *LARF*. In this policy, threads with lower allocation rates are scheduled prior to threads with higher allocation rates. The rationale for proposing this technique is to allow threads that have already completed or are about to complete their allocation phases to manipulate as many objects as possible so that these objects will die. This will allow garbage collection to be more effective in liberating objects, and therefore more heap space will be available for subsequent allocation phases. It is worth noting that this technique still relies on time-quantum for preemption.

The second policy is *Memory-Quantum Round-Robin* or (*MQRR*), a policy that uses heap consumption instead of execution time as the main criterion for thread preemption. MQRR uses *memory-quantum*, which specifies the amount of heap memory that a thread can allocate in each scheduled execution. Once the allocated amount reaches this value, the thread is preempted. The rationale for proposing this policy is to allow threads to make as much progress as possible toward completing their allocation phases. In applications which most threads perform the same task, the memory-quantum can be tuned to closely match the average amount of memory consumed by each allocation phase.

In Section 4 and Section 5, we evaluate the performances of the two proposed techniques against that of a traditional round robin (RR) scheduling policy through trace-driven simulation. We use traces generated from five multithreaded benchmark applications: ECPerf [25], SPECjAppServer2004 [23], SPECjbb2000 [22], hsqldb, and lusearch [5]. These benchmarks are representative of real-world servers and complex multithreaded applications. Our evaluation focuses on three performance areas: garbage collection (mark/cons ratio, pause time), synchronization overheads due to contentions, and overall response time.

In Section 6, we highlight some of the existing issues with the proposed algorithms and our proposed solutions to overcome them. Moreover, we provide a discussion on how to extend the Linux kernels to support the proposed scheduling policies; the discussion includes issues such as how to integrate the proposed algorithms to the dynamic prioritization in Linux and how such an integration makes our algorithms more robust.

2. Motivation

In our previous work, we characterized the lifespans of objects in a SPECjServer2004. We discovered that lifespans become much longer as the number of active threads becomes larger [31]. To better understand the reason for such behavior, we revisit an observation made by Wilson and Moher [28, 30]. They observed that objects are created and die in phases. That is there are phases in which a program allocates a vast amount of objects. In effect, these *allocation phases* set the stage for the *computation-oriented* phases, in which objects created earlier are manipulated and then die. We refer to the amount of heap space needed to complete a phase as the *heap working set*.

When multiple threads share the same heap space, the time-sharing scheduler such as the one used in Linux kernels may interleave the execution of threads in a way that prevent these threads from completing their allocation phases within a mutator interval. Figure 1 provides a simple illustration of such a scenario. The figure assumes that each allocation phase has fixed length and fixed allocation rate (the amount of allocated bytes over time). It also assumes that no objects die in the allocation phases. Figure 1(a) describes the allocation pattern of three threads (T1, T2, T3) ready to run. At that particular time, the heap is empty. The scheduling policy is assumed to be round robin with fixed time quantum.

Each square in Figure 1(b) indicates a quantum that is part of an allocation phase. The scheduler first picks $T1$ to run for one quantum beginning at $Q0$. At the end of $Q0$, $T1$ is suspended, and the scheduler picks $T2$ to run next. Note that in each thread, four quanta are needed to completely allocate the heap working set. These three threads take turns executing and allocating objects in the heap until the beginning of Q8 when the heap is full. At this time, garbage collection is invoked but not one thread has completed its allocation phase. Therefore, none of the objects in the heap can be collected in this example.

Notice that the heap size is large enough to hold two heap working sets (e.g., working sets for T1 and T2). However, interleaved execution prevents each thread from allocating its working set. In our example, no threads are successful, and GC is uselessly invoked. In addition, if the scheduler allows T1 to allocate its heap working set, suspends it, and then allows T2 to do the same, when T3 is scheduled to run,

[1] Our preliminary study indicates that an allocation phase can take as many as 22 quanta to complete in SPECjAppServer2004 [23].

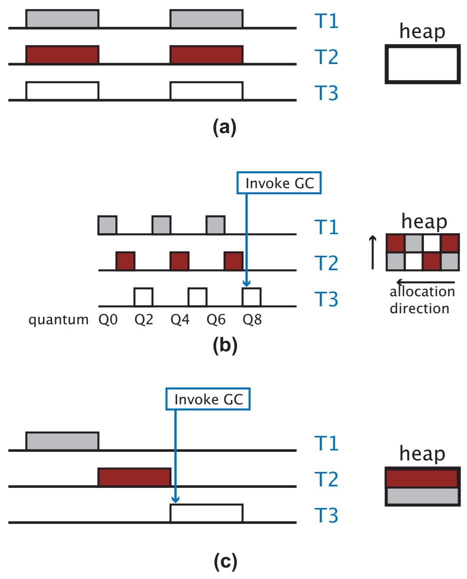


Figure 1. The effect of thread scheduling on garbage collection performance

GC will be invoked because there is not enough room in the heap to satisfy the allocation requests made by T3. Even though the working sets of T1 and T2 are in the heap, both T1 and T2 have not had the opportunity to execute in the computation-oriented phases so most of the objects in the heap are still alive. Again, performing garbage collection in this scenario is useless.

It is also worth noting that if the scheduler were to schedule T1 and then T2 instead of scheduling T3 as shown in Figure 1(c), both T1 and T2 could have completed their computation-oriented phases, and by the time GC is invoked, most objects in the heap would have died. From the object lifespan perspective, inefficient GC due to scheduling can make the lifespans of objects appear to be longer. In fact, we have observed the increasing lifespan phenomenon in our previous study of a SPECjAppServer2004 [32, 31].

This simple example illustrates the influence of thread scheduling on GC performance. In fact, the two scenarios shown in Figure 1(b) and Figure 1(c) can be avoided if the scheduler considers allocation phase behavior and applies the following two principles as part the scheduling process:

1. Give higher execution priority to threads in computation-oriented phases so that more objects will die.
2. Schedule threads in such a way that minimizes the number of partial working sets in the heap.

In the next section, we introduce two scheduling policies that leverage allocation-phase information provided by the JVM to schedule threads so that better GC efficiency can be achieved.

3. Proposed Scheduling Policies

We hypothesize that if thread-scheduling mechanisms are designed to satisfy these two conditions, the performance of garbage collection in multithreaded applications will improve. In this paper, we propose two new scheduling policies. These two policies do not change the way threads are suspended for I/O services or essential operating system interruptions (e.g., signals). However, they change the ways execution-ready threads are scheduled and currently running threads are preempted.

3.1 Lower Allocation Rate First (LARF)

LARF is designed to be easily integrated with existing round robin based scheduling mechanisms. In this approach, the Java Virtual Machine (JVM) maintains an allocation rate (per quantum) of each thread when it was last executed. This information is used to determine the execution priority; threads with lower allocation rates are scheduled ahead of threads with higher allocation rates. Time-based quantum is used to preempt the executing thread.

The major benefit of this technique is that threads that appear to be in the computation-oriented phases have higher execution priorities, satisfying the first condition. However, by using time-based quantum to determine preemption, it becomes more challenging to achieve the second condition as the volume of allocated objects is a more precise metric to describe the heap working set than time.

We will discuss in detail the way LARF is simulated in this work (see Section 4). It is also possible that a thread with extremely high allocation rate may starve since other threads with lower allocation rates always have higher execution priority. We will discuss our strategy to prevent starvation as well as a way to integrate this mechanism to the existing Linux scheduling mechanism (see Section 6).

3.2 Memory-Quantum Round Robin (MQRR)

MQRR is designed to support the two conditions. In this approach, time-based round robin is replaced with memory-based round robin, a policy that regulates the amount of memory each thread can allocate during its turn on the CPU. For example, if the memory quantum is set to 200 KB, a thread can stay on the processor until it allocates 200 KB of heap memory. At that point, the thread is suspended, and the next successive thread is scheduled.

If the memory quantum is tuned to be slightly larger than the most common heap working set, it can ensure that in most cases a thread has enough time on the processor to allocate its current working set (thus, satisfying the second condition) and then execute the subsequent computation-oriented phase (thus, satisfying the first condition). Note that a thread in computation-oriented phase infrequently allocates objects, and therefore, it can stay on the CPU longer, allowing it to "consume" more objects in each execution quantum. The thread is then suspended at the beginning of the next allocation phase. Though the suspension may leave a partial working set in the heap, its size should be small.

Hypothetically, it is possible that MQRR may need more memory quanta to completely allocate a large heap working set than a time-based round robin approach. For example, let's assume that thread $T1$ allocates heap memory at the rate of 4 MB per second. If the memory quantum is set at 200KB based on a common heap working set found in other threads, $T1$ will use up its memory quantum in 50 milliseconds. Let's further assume that $T1$ is trying to allocate a working set of size 2 MB; it will need ten quanta. On the other hand, a time-sharing scheduler with a fixed size quantum of 100 milliseconds can allocate the working set in five quanta.

Based on our preliminary result, such a scenario is unlikely as our study using Linux kernel shows that multiple time-based quanta (ranging from 3 quanta to 22 quanta) are needed for a thread to allocate a heap working set. Because our memory quantum is long enough to allocate a common working set, which is equal to multiple time-based quanta, threads should be able to allocate larger working sets using fewer memory quanta than time quanta.

In the next section, we will discuss the implementation of MQRR in our simulator. The discussion on integrating MQRR with the existing scheduler in Linux is provided in Section 6.

4. Simulation Environment

In spite of many shortcomings such as its inability to provide realtime performance, simulation is still an attractive approach for our experiment because the proposed MQRR is quite complex and will require a significant implementation effort to ensure correct functionality. Simulation also provides us with a common platform to study and compare the performance of all three algorithms (round robin, LARF, and MQRR), while filtering out other runtime factors such as competitions for the CPU time from other threads in the system. In addition, past studies have shown that simulation can provide efficient ways to conduct research in the areas of operating system and garbage collection [24, 13, 21].

Our simulation environment makes the following assumptions. First, we assume that there is only one CPU in the system. This assumption simplifies the simulation environment as there is only one *ready queue* instead of one for every processor in the simulator. Second, we ignore I/O events as they normally cause threads that request I/O services to be suspended. Thus, these threads are in the blocking state, which is not affected by our proposed scheduling policies. Third, we assume that the execution flow of each thread does not change after we apply different scheduling strategies to reorder the execution sequence. This assumption can guarantee that the heap mutation sequence of a thread is not affected by scheduling strategies. Additionally, our simulator assumes that semaphore is used for locking objects. Many JVMs today, including HotSpot, utilize thin-lock as a way to reduce the synchronization overhead [26]. Thin lock uses hardware instructions such as compare-and-swap to guard objects shared by multiple threads. The first time contention occurs in a shared object, spin-lock is used to prevent data races. Afterward, traditional locking mechanisms such as semaphore are used to lock that particular object [3]. In MQRR, the use of spin-lock can cause our simulator to become live-lock since the main preemption criterion is the volume of heap allocation. Thus, when a contention occurs in our simulation, we assume that the thread attempting to access the locked object will be blocked.

The input to our simulator is the runtime trace of each thread. We instrumented Sun HotSpot VM to capture all the allocation events. We utilized Merlin algorithm [11] to efficiently and precisely compute the object reachability information that can be used to derive the lifespan information. To track the synchronization behavior, we recorded all *monitor enter* and *monitor exit* events during the execution as they are commonly used to synchronize objects. We also recorded the identifications of threads that access each shared object. We also placed a timestamp after each event that can be used for event synchronization during simulation.

The remaining configurable parameters include scheduling strategy (MQRR, LARF, or round robin), heap organization (e.g., heap size, ratio between minor and mature generations), and garbage collection techniques (mark-compact, semi-space copying, and two generational collectors). The outputs of the simulator are metrics that describe the garbage collection performance (e.g., number of GC invocations and mark/cons ratio) and the overall performance (e.g., context-switching events and synchronization overhead).

4.1 Simulating Thread Scheduling

Figure 2 provides an overview of our scheduling simulator. First, our simulator initializes all threads in the *ready queue* in the order of their creation times. Then it schedules the very first thread in the ready queue and simulates its execution based on the desired policy (e.g. RR, MQRR) by reading its trace information. If a *monitor_enter* event is encountered, the simulator checks whether the thread is trying to acquire a lock already owned by some other thread. If so, the thread is placed at the end of the *waiting queue*; otherwise the simulator continues to execute the thread. If the simulator encounters a *monitor_exit* event, a lock related to the monitor is released, and all threads competing for that lock are moved from the waiting queue to the ready queue. When a quantum (time-based or memory-based) runs out, the thread is put on the *ready queue*, and the scheduler picks a successor thread from the *ready queue* as specified by the scheduling strategy.

As the mutation sequence changes due to different scheduling policies, there are several challenges that must be addressed:

- **Determining thread creation time.** In our simulation, execution flows are different when different scheduling

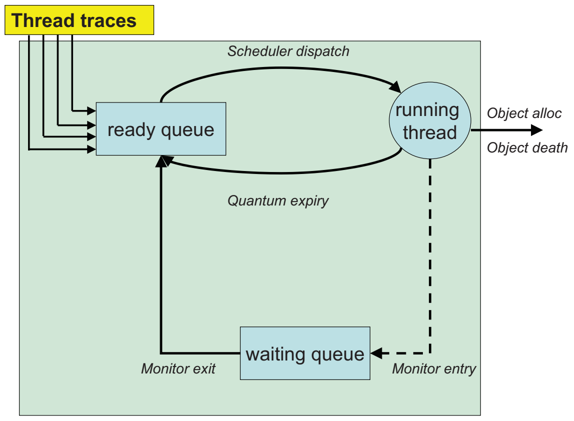


Figure 2. Overview of our schedulers simulator

strategies are used. Therefore, we use a time-insensitive approach to determine when a thread is created. In this method, we use the allocated bytes of the *main* thread as a reference to record each thread's creation time. In our simulation, $T1$ is created when the *main* thread has allocated X bytes. This is reasonable because the *main* thread spawns most, if not all, threads in an application.

- **Detecting quantum expiry.** Assuming that a CPU quantum is T seconds, when a thread is scheduled to run, our simulator records the timestamp of the first event, which is denoted as $T0$. As the simulator processes the subsequent events of that thread, it also checks the timestamp of each event; if the timestamp of an event is greater than T+T0, then the thread has used up its quantum. The thread is then inserted into the *ready queue*. Otherwise, the simulator continues to execute the thread.
- **Taking synchronization into account.** To simulate monitor events, we set up a hash table to record every object's basic information, such as its identification and its size. We also associate each object with a sync-lock, which is used to identify if this object is being accessed by another thread. At every object allocation event, our simulator adds a new entry to the hash table. When a thread tries to access a shared object, our simulator first checks the sync-lock. If the object is in use, the thread is added to the waiting queue.

4.2 Simulating GC Behavior

We used our simulator to study the effect of our technique on four different "stop-the-world" garbage collection algorithms:

Mark-compact (MarkCompact) collects the heap in two phases: the first is marking all live objects; the second phase is compacting the heap by moving all live objects into contiguous memory locations.

Semi-space copying (SemiSpace) divides the heap equally into two copy spaces. It allocates in the *from* space, and once this space is full, copies surviving objects to the *to* space. Once the garbage collection process has completed, the two labels, *from* and *to* are swapped.

Generational copying (GenCopy) utilizes two spaces in the heap: a nursery space, which contains all newly allocated objects since the last GC, and a mature space, which contains the remaining objects. When the nursery space is full, a minor collection is performed and all surviving objects are promoted into the mature space. The mature collector uses SemiSpace collector.

Generational mark-compact (GenMC) uses copying in the nursery space and MarkCompact in the old space. This technique is used in many commercial virtual machines including HotSpot [26] and .NET Framework [17].

For more information, please refer to Wilson's survey paper of uniprocessor garbage collection [29] or Jones and Lin's book on garbage collection [15].

5. Evaluation

In this section, we evaluate the performances of LARF and MQRR against that of round robin (RR), a widely adopted strategy in time-sharing schedulers. Our evaluation includes the effect of our proposed algorithms on the GC performance and the overall performance using GenMC as the default collection algorithm. We chose GenMC because this technique has been widely used in many commercial virtual machines [26, 17]. We set the heap size in our experiment to be three times larger than the maximum live size as this value yields a reasonable performance for generational garbage collector [10]. We also configured the size of the mature space to be twice as large as that of the nursery space. Our previous study showed that this value yields the best performance for multithreaded server applications [32, 31]. We also evaluated the sensitivities of our algorithms to heap sizes and garbage collection policies.

The platform used for trace generation and simulation was an AMD Athlon workstation running Linux 2.6. In our simulation, the CPU quantum of RR and LARF was set to $1.14 * 10^{-3}$ seconds, the average quantum length of our platform. Note that our study indicates that threads often get preempted prior to quantum expiry, and thus they spend only about $1.14 * 10^{-3}$ seconds on the CPU. Because our trace generator filters out I/O accesses and page faults, giving a full quantum (e.g., 10 0 ms) to each simulated thread may not be representative of real-world systems.

The memory-quantum of MQRR is set to 10 KB. We adopted this value for two reasons. First, our preliminary investigation of the allocation phases showed that most phases have a working set of about 10 KB. We also conducted many experiments using multiple memory-quantum values and discovered that 10KB yielded consistently good results in all benchmarks.

Benchmark	Description	Input configurations	Total allocations		Maximum live size (MB)	Number of threads
			objects(million)	bytes(MB)		
hsqldb	Executes a number of transactions against a model of a banking application	-s default	4.43	134.36	80.11	81
lusearch	Performs a text search of keywords over a corpus of literature data	-s default	16.4	2101.92	3.95	32
SPECjbb2000	A Java program emulating 3-tier system focusing on the middle tier	8 warehouses	1113.86	128161.5	145.52	36
SPECjAppServer2004	A J2EE benchmark emulating an automobile manufacture company	transaction rate = 1	48.761	1501.42	116.01	407
ECPerf	An original version of jAppServer2004 but provides different workload (no web tier)	transaction rate = 1	34.112	1128.01	101.12	405

Table 1. Benchmark Characteristics

5.1 Benchmarks

We chose two benchmarks: *hsqldb* and *lusearch* from *DaCapo* suites [5][2]. We needed to subset the DaCapo suite because most of the benchmarks are not multithreaded. The remaining three benchmarks are *SPECjbb2000, SPECjAppServer2004*, and *ECPerf*. Table 1 shows the brief description and characteristics of these five benchmarks.

5.2 Garbage Collection Performance

We use *mark/cons* ratio [4, 7, 11] to measure the GC overhead. Mark/cons ratio is defined as the total number of bytes copied during all garbage collections divided by the total number of allocated bytes. The metric indicates the GC work per allocated byte. Work by Hirzel et al. [13] also uses mark/cons ratio to evaluate the simulated performance of a garbage collection strategy.

Figure 3(a) shows the mark/cons ratio of GenMC under RR, MQRR, and LARF scheduling strategies. In the graph, the higher bars indicate worse performance. Table 2 gives the number of garbage collection invocations under the three strategies. It is worth noting that the mark/cons ratio of *hsqldb* is not affected by scheduling strategies. *Hsqldb* first loads a large database (about 80MB) into the heap, and then generates 80 threads to query the database. Each thread performs several SQL operations, which are very short. Also the allocation size of each thread is less than 1MB. We observed that all these threads die before they encounter their first GCs regardless of the scheduling strategies.

For *lusearch*, LARF and MQRR show a 10% reduction of mark/cons ratio. For the remaining benchmarks, LARF and MQRR can reduce the mark/cons ratio by 15% in *SPECjbb2000*) to 25% in *SPECjAppServer2004*. We can achieve such reductions because LARF and MQRR give higher priority to threads in computation-oriented phases. Therefore, there are more dead objects at each GC invocation point.

5.3 Pause Times

In stop-the-world garbage collectors, the amount of garbage collection overhead determines the pause time of a program in two aspects: pause due to each GC invocation, which reflects the disruptiveness of the whole program, and pause time per thread, which reflects the GC stoppage within the execution of each thread.

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

5.3.1 Pauses due to each GC invocation

We measured pause time per GC as the amount of copying work done by each GC invocation divided by the heap size. Figures 3(b) and 3(c) depict the average and maximal pause time per GC, respectively. The graphs indicate that LARF and MQRR can significantly reduce the pause time of each GC in four out of five benchmarks. Again, *hsqldb* is not affected by the different scheduling policies. The result also indicates that the two scheduling algorithms allow garbage collection to be more efficient.

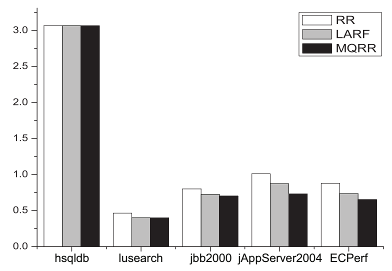
5.3.2 GC pause time per thread

In stop-the-world garbage collectors, all threads are stopped during a garbage collection invocation. To investigate the effect of garbage collection on each thread, we measured the time spent by each thread doing the GC work divided by its allocation size. This metric partially reflects the mutator utilization of each thread. Simply, a higher pause time indicates a lower mutator utilization by a thread.

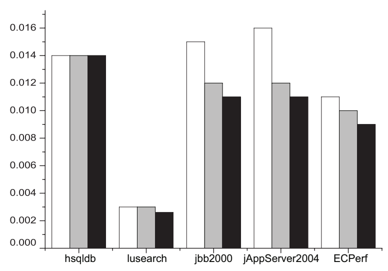
Figure 4 illustrates the boxplots of the pause times of the five benchmarks. Each boxplot can be interpreted as follows: the box contains the middle 50% of the data from the 75th percentile of the data set (represented by the upper edge of the box) to the 25th percentile (represented by the lower edge); the line in the box means the median value of the data set; the whiskers at both ends of the vertical line indicate the minimum and maximum values.

Once again, the boxplot of *hsqldb* using all three scheduling strategies show no differences in performance. It can be seen that most threads experience no pauses, meaning that there were no GC invocations during their lifetimes.

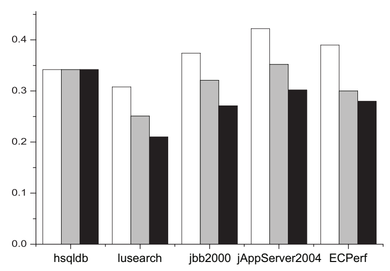
The compactness of boxplots indicates the fairness of scheduling strategy. The term fairness means that threads allocating fewer objects should spend less time waiting for GC to be completed. As shown in Figure 4, the boxplots of MQRR are tighter than those of RR and LARF. This is because threads that are less active (i.e., threads that allocate fewer objects) experience shorter GC pauses in MQRR since they are scheduled earlier than when RR and LARF are used.



(a) Mark/cons ratio



(b) Average pause time/GC



(b) Maximum pause time/GC

Figure 3. Illustrations of mark/cons ratio and the pause times per GC

Benchmark	RR		LARF		MQRR	
	Minor	Full	Minor	Full	Minor	Full
hsqldb	6	5	6	5	6	5
lusearch	2673	2	2578	2	2301	2
SPECjbb2000	1521	35	1301	25	1290	25
SPECjAppServer2004	191	94	180	78	162	65
ECPerf	123	84	93	76	102	60

Table 2. Number of garbage collection invocations

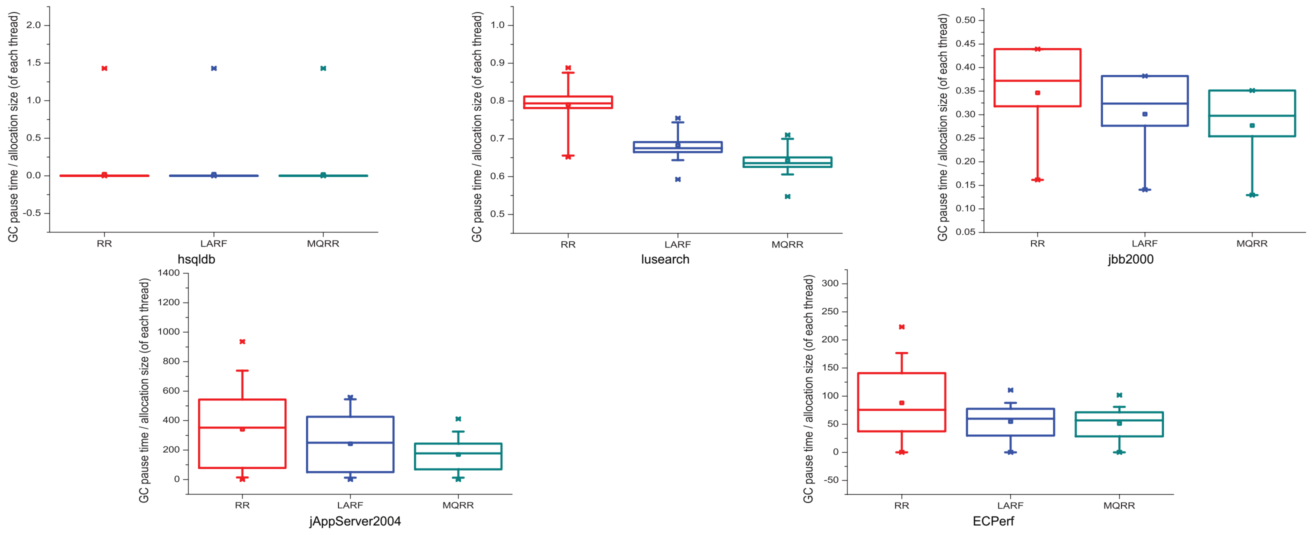


Figure 4. Comparing the GC pause time per thread

5.4 Overall Performance

Other than garbage collection time, scheduling strategies can affect synchronization time and context-switching time, which in turn, affect the turnaround time of an application.

5.4.1 Synchronization overhead

Different scheduling strategies yield different sequences of execution. The changes in execution sequence can affect the synchronization behavior, resulting in different synchronization overheads. As we stated earlier, thread synchronization is mainly implemented by monitors in Java. There are two kinds of monitor-related events, which incur overhead:

- *Entering and exiting never contended monitors*. When a thread enters a monitor, it will attempt to acquire a thin-lock associated with the monitor [3]. It performs a Compare-And-Swap (CAS) operation to check if the lock has been set by other thread. Thus, the major cost of entering a monitor is execution cost of the CAS instruction.
- *Entering and exiting contended monitors*. A monitor contention occurs when a thread attempts to enter a moni-

tor already acquired by another thread. In JVMs utilizing thin-lock mechanism, the first time the contention occurs, a spin-lock mechanism is used to hold the thread at the entry point. From this point on, heavyweight lock will be used to synchronize this object. Thus, the major cost of the subsequent contentions is the time spent on system calls to acquire and/or manipulate the heavyweight lock.

The major synchronization overhead is mainly due to contentions. Thus, we evaluate the synchronization overhead as the number of monitor contentions with respect to each scheduling mechanism. Table 3 reports our simulation results. LARF has more synchronization contentions than RR in *hsqldb* and *ECPerf* but the increase is less than 10%. MQRR experiences more synchronization contentions than RR in *lusearch* and *jServer04*. In the worst case, the increase in the number of contentions due to MQRR is 15% in *lusearch*.

5.4.2 Context-switching overhead

Table 3 reports the number of context-switching events when different scheduling strategies are used. As shown in the table, LARF has a greater frequency of context-switching than RR in *hsqldb* and *ECPerf*. This is mainly due to the increasing monitor contentions. On the contrary, the number of context-switching events drops significantly in MQRR because the memory-quantum in MQRR often spans the entire allocation phase, which generally consists of several CPU time slices in RR. Therefore, threads are suspended less frequently in MQRR when compared to RR and LARF.

5.4.3 Total execution time

To calculate the overall execution time of a multi-threaded program, we need to add up the execution times of all threads, the total GC time, and any other overheads, including the synchronization and context-switching. We use the following formula to calculate the total execution time.

$$Total_{exec} = \sum_{i=1}^{n}(T_{exec_i}) + c_{gc} * V_{gc} + c_{syn} * N_{syn} + c_{cs} * N_{cs}$$

In this formula, T_{exec_i} is the execution time of the *i*th thread; c_{gc} is the average marking/copying time per byte; V_{gc} is the total GC work in bytes, which is indicated in Section 5.2. Parameter c_{syn} is the average cost of each monitor contention, and N_{syn} is the number of monitor contention events. Parameter c_{cs} is the average time of a context-switching event, and N_{cs} is the number of context-switching events.

Note that parameters c_{gc}, c_{syn} and c_{cs} are highly dependent on the underlying OS and architecture. For our evaluation, we conducted experiments to identify the average values of these parameters. Our experiments yield the following values: $c_{gc} = 1.8 * 10^{-8}$ seconds, $c_{syn} = 2.7 * 10^{-6}$ seconds and $c_{cs} = 2.3 * 10^{-9}$ seconds.

Figure 5(a) depicts the reduction of total execution time (compared to RR) of benchmarks when LARF and MQRR are used. The result shows that the total execution time is reduced by about 3% due to the decreasing GC time.

5.4.4 Average turnaround time

One important metric that has commonly been used to evaluate scheduling strategies is the average turnaround time of threads. We calculated the turnaround time by summing up the execution times, suspended times (due to preemption and execution of other threads), the GC time, the synchronization and the context-switching overheads during an application's lifetime. The same parameters are used to describe the GC time (c_{gc}), the monitor contention cost (c_{syn}), and the context-switching time (c_{cs}) as used in the previous section.

Figure 5(b) depicts the reductions of the average turnaround times (compared to RR) of all benchmarks when LARF and MQRR are used. The results show that LARF can reduce the average thread turnaround time by up to 12%. MQRR performs slightly better than LARF, in which the average turnaround time is reduced by 13%.

5.5 Sensitivity to Different Garbage Collection Techniques

We evaluated the scheduling strategies under other commonly used garbage collection techniques: SemiSpace, MarkCompact, and GenCopy. In our experiment, we used RR as the baseline strategy. For each scheduling strategy (MQRR or LARF), we measured its GC time, the total execution time, and the average thread turnaround time of each GC technique.

Figure 6 and 7 illustrate the results of MQRR and LARF, respectively. To simplify the comparison, we reported our results based on reduction ratios as compared to the performance of RR. We included the results of GenMC in the graphs. LARF and MQRR also showed performance improvement in GenCopy. The improvement is better than GenMC in most benchmarks.

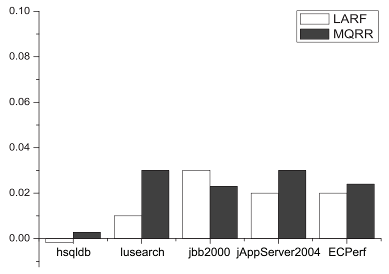
Interestingly, MQRR and LARF yielded very little performance improvement over RR when used with SemiSpace and MarkCompact collectors. In the worst case, LARF increases the GC time by 15% in *ECPerf* when using MarkCompact collector. In these two collectors, the mutation time between two consecutive GCs is generally longer than the mutation time in generational collectors due to smaller nursery space. We believe that longer mutation intervals neutralize the benefit of our scheduling techniques.

5.6 Sensitivity to Heap Size

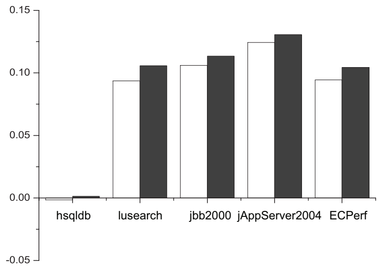
Table 4 shows the results of LARF and MQRR when the heap is set to 1.5 times larger than the live-size. The results show a similar performance reduction of GC time (ranging from 2% to 21% and 5% to 28% for LARF and MQRR, respectively). Notice that both LARF and MQRR perform better because under tight heap condition, the mutator inter-

Benchmark	Monitor contentions				
	hsqldb	lusearch	SPECjbb2000	SPECjAppServer2004	ECPerf
RR	3051	216	85	81004	67213
LARF	3266	208	82	80234	68498
MQRR	2991	250	79	85246	64539
	# of context-switching events				
	hsqldb	lusearch	SPECjbb2000	SPECjAppServer2004	ECPerf
RR	5907	1.35 million	23.10 million	5.41 billion	3.21 billion
LARF	6895	1.29 million	23.10 million	5.41 billion	3.32 billion
MQRR	3911	1.12 million	20.34 million	4.30 billion	3.07 billion

Table 3. Number of synchronization contentions and context switching events

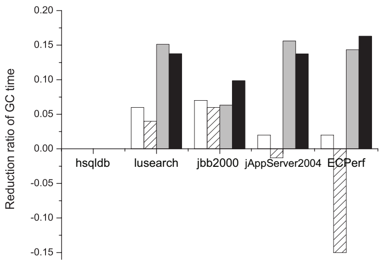


(a) Execution time

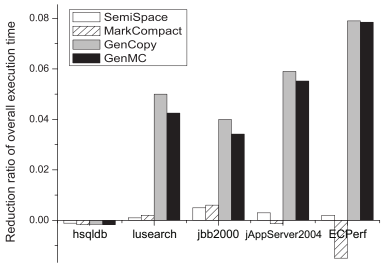


(b) Turn-around time

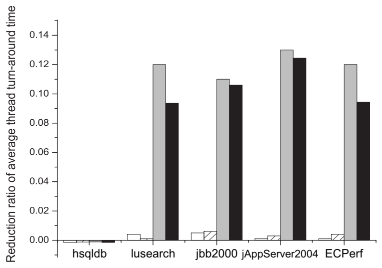
Figure 5. The reduction of total execution time (a) and turnaround time (b) in LARF and MQRR (relative to RR)



(a) GC time



(b) Execution time



(c) Turn-around time

Figure 6. The reduction percentages of GC time, total execution time, and average turnaround time in LARF (with respect to RR) using four collectors

vals are shorter, meaning that there are more opportunities for savings. For LARF, the reduction can range from 1% to 4%. For MQRR, the reduction is from 2%-6%. It is worth noticing that LARF and MQRR can further reduce average turnaround time under the tight heap size, compared to 3X heap size.

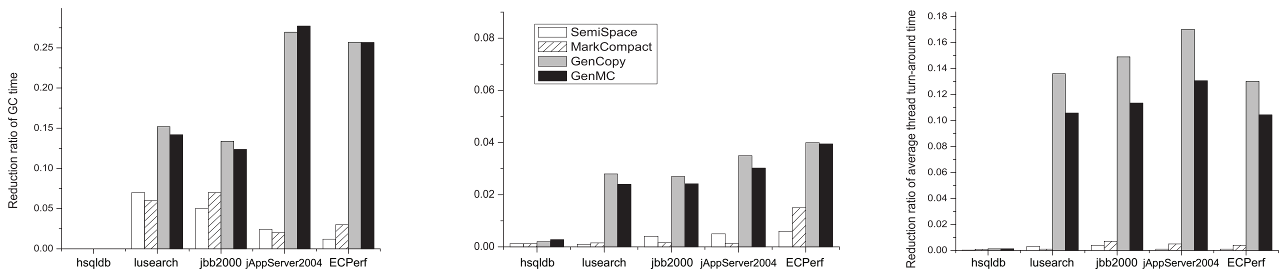
6. Discussion

In this section, we discuss some of the runtime issues with the proposed scheduling algorithms. We also provide relevant background related to the scheduling mechanism in Linux kernel version 2.6 as well as the plan to integrate our algorithms to Linux.

6.1 Issues to Be Resolved

Starvation. When the proposed LARF is utilized, it is possible that starvation can occur as the scheduler prioritizes threads with lower allocation rates. This problem can be alleviated by using a prioritization mechanism similar to that used by Linux (see Section 6.3).

Live-lock. When MQRR is used, a thread in busy waiting loop may stay on the processor forever. This situation is referred to as live-lock and can occur because the thread is not likely to allocate any objects while in this loop; thus, it will never use up its memory quantum and be suspended. We



(a) GC time (b) Execution time (c) Turn-around time

Figure 7. The reduction percentages of GC time, total execution time and average turnaround time in MQRR (with respect to RR) using four collectors

	LARF						MQRR					
	GC time (secs)	Reduction relative to RR (%)	Exec. time (secs)	Reduction relative to RR (%)	Average turnaround time (secs)	Reduction relative to RR (%)	GC time (secs)	Reduction relative to RR (%)	Exec. time (secs)	Reduction relative to RR (%)	Average turnaround time (secs)	Reduction relative to RR (%)
hsqldb	3.83	**2.54**	9.09	**1.09**	2	**0**	3.73	**5.09**	8.99	**2.18**	2	**0**
lusearch	4.06	**14.71**	18.50	**3.65**	10	**9.09**	3.86	**18.91**	18.30	**4.69**	9.30	**15.45**
SPECjbb2000	159	**13.11**	1170	**2.01**	102	**13.56**	141	**22.95**	1152	**3.52**	97	**17.80**
SPECjAppServer2004	504	**16.14**	2914	**3.22**	183	**17.19**	501	**16.64**	2911	**3.32**	174	**21.27**
ECPerf	322	**21.46**	1914	**4.40**	181	**11.27**	292	**28.78**	1884	**5.89**	173	**15.20**

Table 4. Performance improvement of LARF and MQRR when the heap size of 1.5 times of the maximum live-size and GenMC is used

plan to use a time-out mechanism as a back-up preemption policy to prevent live-lock from occurring.

Deployment in general-purpose systems. A computer system usually has many applications running simultaneously; some utilize garbage collection, and some do not. Because our proposed algorithms are mainly for applications that employ garbage collection, it is unclear how the two algorithms will perform in systems with many applications not utilizing garbage collection (e.g., application written in C). This issue will be discussed in the next section as part of the plan to integrate the proposed policies into Linux kernels.

6.2 Thread Scheduling in Linux

Linux adopts a scheduling policy that categorizes tasks into compute-bound (not to be confused with the term *computation-bound* introduced by Wilson and Moher [28, 30]) and I/O-bound. Compute-bound tasks rarely sleep and rarely get suspended to perform I/O operations. On the other hand, I/O-bound tasks spend a large amount of time sleeping or blocking on I/O operations. To make sure that the compute-bound tasks are not unfairly utilizing the CPU, a dynamic task prioritization mechanism is used to lower the priorities of compute-bound tasks, ensuring that other tasks also get time to execute on the CPUs [1].

Beginning in the kernel version 2.6, two priority arrays (arrays of linked lists) are used to provide constant-time thread management overhead. Each array has 140 elements representing priority levels; only one array is active at a time. Tasks are scheduled based on the order of priority, and within each priority level, tasks are scheduled in a round robin fashion. When an executing thread has used up its quantum, a new priority is calculated by subtracting the time the task spent executing from the time the task spent on sleeping or blocking. Once the new priority is determined, the task is added to the corresponding linked list in the inactive priority array. When there are no more tasks in the active priority array, pointers to active and inactive priority arrays are swapped [1].

6.3 Integration Plan

While we can only report the results based on simulation, we have already developed a plan to implement the proposed LARF and MQRR into Linux kernels. To incorporate LARF, the first step is to create a system call that can be invoked by HotSpot to record the allocation rate from the latest execution quantum of each thread. The information will be stored in the existing data structure that maintains thread information (i.e., *task_struct*). We can implement LARF by simply modifying the function that dynamically determines the thread priority by using the allocation rate in addition to the sleeping time and execution time. In doing so, we can avoid starvation as each thread will get a chance to execute.

MQRR is more challenging to implement than LARF because it no longer relies on time-based quantum. Because each object allocation in the heap usually does not require operating system support (the exceptions are when the system needs to commit more memory, the memory access incurs page fault, or the heap needs to be enlarged, etc.), the operating system may not be fully aware of the amount of allocated objects in the heap. Our current plan is to create a software interrupt that can be invoked by the dynamic memory allocator in the JVM to notify the operating system that the executing thread has used up its memory-quantum. The existing algorithm that uses sleeping time and execution time to calculate priority will also be used by our system. The same mechanism is still applicable because recently suspended threads are likely to be in the middle of allocation phases; therefore, should have lower priorities.

Because we plan to extend the existing scheduling mechanism in Linux to support the proposed policies, LARF and MQRR can coexist with the default scheduling policy in Linux. This coexistence will allow us to selectively apply our algorithms to Java threads, while continuing to use the default scheduling policy for non-Java threads. To utilize LARF, modifications must be made to the function that determines priority and not the priority arrays. Thus, once Java threads are added to the priority array, they can be scheduled in a similar fashion to non-Java threads. On the other hand, we may need to extend the priority arrays to support MQRR so that both time-base quantum and memory-based quantum can be used. One solution is to have two linked lists for each priority, one for non-Java threads and the other for Java threads.

7. Related Work

Operating systems have played an important role in improving the performance of garbage collection. For example, virtual memory protection mechanisms have been used to reduce the overhead of write-barriers, a common procedure to track reference assignments [2, 6, 20]. In addition, recent research efforts by Yang et al. [33] and Grzegorczyk et al. [8] leverage information from the operating system to maximally set the heap size while minimizing the paging efforts.

Information made available by the operating system has also been used to explain performance issues and identify memory errors. Work by Hauswirth et al. [9] uses information from operating systems as well as other software and hardware layers to understand performance. One of their examples investigates the effect of paging on GC performance. A study by Hibino et al. [12] investigates the differences in the performance degradation of Java Servlets among operating systems.

To the best of our knowledge, there have not been any research efforts to create specialized schedulers to improve garbage collection performance. However, there have been several efforts that make scheduling decisions based on the resource availability. Such schedulers are referred to as resource-aware scheduling. *Capriccio*, a system introduced by von Behren et al. [27] makes scheduling decisions based on resource usage to avoid resource thrashing. Work by Philbin et al. [19] also discovers that execution order can affect cache locality. Their work introduces a scheduling algorithm aiming at reducing cache misses.

Narlikar [18] introduces *DFDeques*, a space efficient and cache conscious scheduling algorithm for parallel programs. Multiple ready queues are globally organized to fully take advantage of available parallelism. For example, if a ready queue belonging to a processor is empty, the scheduler can assign a task from another ready queue to gain more parallelism. The scheduler also applies *memory threshold*, which limits the amount of memory a processor may allocate when consecutively executing jobs from other ready queues. If a processor has exhausted its memory quantum, the executing thread is suspended. The similarity between this work and our work is that memory consumption is used as a criterion for thread preemption.

8. Conclusion

In this paper, we introduce two new scheduling strategies, MQRR (memory-quantum round robin) and LARF (lower allocation rate first), designed to be allocation-phase aware. Both schemes assign higher execution priority to threads in computation-oriented phases, allowing more objects to die. The results of our simulation indicate that the two schemes perform better when generational schemes are used. However, they do not perform well when non-generational collectors are used.

Compared to round robin, the reductions of the garbage collection time of can range from 0%-16% and 0%-27% when LARF and MQRR are used, respectively. The reductions of the overall execution time range from -0.1%-3% for both LARF and MQRR. The reductions of the average thread turnaround time range from -0.1%-12% for LARF and 0.1%-13% for MQRR.

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] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 96–107, Santa Clara, California, USA, 1991.

[3] 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.

[4] H. G. Baker. Infant mortality and generational garbage collection. *SIGPLAN Not.*, 28(4):55–57, 1993.

[5] 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, Oregon, USA, 2006.

[6] H. J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 157–164, Toronto, Ontario, Canada, 1991.

[7] W. D. Clinger and L. T. Hansen. Generational garbage collection and the radioactive decay model. *ACM SIGPLAN Notices*, 32:97–108, 1997.

[8] 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, USA, March 2007.

[9] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical Profiling: Understanding the Behavior of Object-Oriented Applications. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 251–269, Vancouver, British Columbia, Canada, October 2004.

[10] M. Hertz and E. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 313–326, San Diego, CA, USA, October 2005.

[11] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-Free Garbage Collection Traces: How to Cheat and Not Get Caught. In *Proceedings of the 2002 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 140–151, Marina Del Rey, California, 2002.

[12] H. Hibino, K. Kourai, and S. Shiba. Difference of Degradation Schemes among Operating Systems: Experimental Analysis for Web Application Servers. In *Workshop on Dependable Software, Tools and Methods*, Yokohama, Japan, July 2005. http://www.csg.is.titech.ac.jp/paper/hibino-dsn2005.pdf.

[13] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 359–373., October 2003.

[14] JBoss. Jboss Application Server. Product Literature, Last Retrieved: June 2007. http://www.jboss.org/products/jbossas.

[15] R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic Dynamic Memory Management*. John Wiley and Sons, 1998.

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

[17] Microsoft. About the Common Language Runtime (CLR). http://www.gotdotnet.com/team/clr/about_clr.aspx.

[18] G. J. Narlikar. Scheduling threads for low space requirement and good locality. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 83–95, Saint Malo, France, 1999.

[19] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. *SIGOPS Operating Systems Review*, 30(5):60–71, 1996.

[20] R. A. Shaw. *Empirical analysis of a LISP system*. PhD thesis, Stanford University, Stanford, CA, USA, 1988.

[21] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: Simple and effective adaptive page replacement. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 122–133, Atlanta, Georgia, United States, 1999.

[22] Standard Performance Evaluation Corporation. SPECjbb2000, 2000. http://www.spec.org/osg/jbb2000/docs/whitepaper.html.

[23] Standard Performance Evaluation Corporation. SPECjAppServer2004 User's Guide. On-Line User's Guide, 2004. http://www.spec.org/osg/jAppServer2004/docs/UserGuide.html.

[24] D. Stefanović, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 370–381, Denver, Colorado, United States, November 1999.

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

[26] Sun Microsystems. Performance Documentation for the Java HotSpot VM. On-Line Documentation, Last Retrieved: June 2007. http://java.sun.com/docs/hotspot/.

[27] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 268–281, Bolton Landing, NY, USA, 2003.

[28] P. R. Wilson. Opportunistic garbage collection. *ACM SIGPLAN Notices*, 23(12):98–102, 1988.

[29] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management (IWMM)*, pages 1–42, St. Malo, France, September 1992.

[30] P. R. Wilson and T. G. Moher. Design of the opportunistic garbage collector. *ACM SIGPLAN Notices*, 24:23–35, 1989.

[31] F. Xian, W. Srisa-an, C. Jia, and H. Jiang. AS-GC: An Efficient Generational Garbage Collector for Java Application Servers. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*, pages 126–150, Berlin, Germany, July 2007.

[32] F. Xian, W. Srisa-an, and H. Jiang. Investigating the throughput degradation behavior of Java application servers: A view from inside the virtual machine. In *Proceedings of the 4th International Conference on Principles and Practices of Programming in Java*, pages 40–49, Mannheim, Germany, 2006.

[33] 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.