Exploiting FIFO Scheduler to Improve Parallel Garbage Collection Performance

Junjie Qian [§], Witawas Srisa-an [§], Sharad Seth [§], Hong Jiang [§ †], Du Li [‡], Pan Yi [§]

[§]University of Nebraska Lincoln, [†]University of Texas Arlington, [‡]Carnegie Mellon University

{jqian, witty, seth, jiang, pyi}@cse.unl.edu, [‡]dawn2004@gmail.com

Abstract

Recent studies have found that parallel garbage collection performs worse with more CPUs and more collector threads. As part of this work, we further investigate this phenomenon and find that poor scalability is worst in highly scalable Java applications. Our investigation to find the causes clearly reveals that efficient multi-threading in an application can prolong the average object lifespan, which results in less effective garbage collection. We also find that prolonging lifespan is the direct result of Linux's Completely Fair Scheduler due to its round-robin like behavior that can increase the heap contention between the application threads. Instead, if we use pseudo first-in-first-out to schedule application threads in large multicore systems, the garbage collection scalability is significantly improved while the time spent in garbage collection is reduced by as much as 21%. The average execution time of the 24 Java applications used in our study is also reduced by 11%. Based on this observation, we propose two approaches to optimally select scheduling policies based on application scalability profile. Our first approach uses the profile information from one execution to tune the subsequent executions. Our second approach dynamically collects profile information and performs policy selection during execution.

1. Introduction

As the speed of microprocessors tails off, utilizing multiple processing cores per chip is becoming a common way for developers to achieve higher performance. To do this, developers shift from writing sequential code to employing multi-threading to achieve execution parallelism by (i) evenly distributing the workload among all threads within the application [6] and (ii) minimizing the number of synchronization operations to promote higher execution parallelism among the threads. As such, many programming languages including Java, support threading as a language feature.

Unfortunately, effective workload distribution, alone, does not guarantee scalability because Java is a managed language and part of the overall execution time is spent on runtime systems such as garbage collection (*GC*), which usually exhibit a very different execution behavior from that of the actual program. Furthermore, studies have shown that garbage collection can take up to one-third of total execution time of an application [9, 26]. As such, any scalability study must decompose the overall execution time into at least two components: time spent in *executing the application* or *mutator time* and time spent in *performing garbage collection* or *GC time*.

Although the Java virtual machine (JVM) creates threads, it is the operating system that schedules them for execution. In JDK1.7, each thread in the JVM is mapped as one kernel thread in Linux and they are, by default, scheduled using Completely Fair Scheduler, which attempts to evenly divide the CPU time to all the processes in the running queue. Though the fairness between all the threads in an application is achieved, this scheduling policy also has adverse effects on garbage collection performance. Specifically, our investigation found that the garbage collector performs worse if the application's workload is evenly distributed among threads. A large number of concurrent threads create heap allocation competition that can lead to prolonged object lifespans. This phenomenon occurs because one thread is preempted before it can fully use the objects it allocated. As a result, although the objects of this thread would die soon, the heap becomes full and the GC is called as other threads compete to allocate objects on the heap. The lower GC efficiency makes the collector copy more objects from young to mature generation, which degrades the overall GC performance.

To the best of our knowledge, there has not been previous work that investigate the impact of scheduling policies on performance of multi-threaded Java applications. The key research question we want to investigate is: *how can different scheduling policies affect GC and application performances?* To answer this question, we conducted a series

VEE '16 April 2–3, 2016, Atlanta, Georgia, USA.
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3947-6/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2892242.2892248

of investigations that make the following research contributions:

- We show that the completely fair scheduler in Linux prolongs the average object lifespan of scalable Java applications. The average longer object lifespan results in less efficient GC, which leads to poorer performance of both GC and whole application.
- We employ the first-in-first-out scheduler, instead, for the multi-threaded Java applications, which avoids allocating objects from all threads on the heap at the same time. Both GC and mutator performances show significant improvements with the first-in-first-out scheduler. Also, the GC scalability of the scalable applications is improved.
- We propose two approaches to select the best performing scheduling policy for an application based on its workload distribution profile. The first approach collects runtime information during a profile run and then selects a suitable scheduling policy based on the collected information for the subsequent runs. The second approach is to dynamically select an appropriate scheduling policy using on-the-fly profiling results.

The rest of this paper is organized as follows. Section 2 provides an overview of process schedulers and garbage collection. Section 3 explains the experimental methodology used in our investigation and reports the impact of the CFS scheduler on the applications performance. Section 4 presents the performance evaluations when using the first-in-first-out scheduler for the Java applications. Section 5 demonstrates two approaches we developed to automatically select an optimal scheduling policy for an application. Section 6 discusses the published works related to this topic. Section 7 discusses future work and concludes this paper.

2. Background

This section provides some background information on the process scheduling techniques in Linux and parallel garbage collection in Oracle OpenJDK 7.

2.1 Process Scheduling

During a life time of a process, it can be in many states that include ready, running, and blocked. When a process is created, it begins its life in the ready state waiting for its turn to execute. Once the process is scheduled, it enters the running state. Later, it may be swapped out as it waits for information from I/O devices or go back to ready state again if it is preempted due to quantum expiration. The number of threads that can run at the same time is often determined by the number of CPUs. The scheduler decides when to run which threads, and the order of the threads' execution has been shown to affect the overall performance of an application [19, 25, 27].

In Linux, at least two scheduling algorithms, *Completely Fair Scheduler* (*CFS*) and *First-In-First-Out* (*FIFO*) scheduler, are available. Within a system, *it is possible for one or more applications to use FIFO while the rest of the applications use CFS.*

Completely Fair Scheduler. CFS is currently the default scheduler in Linux [2] for system-level scheduling. Its goal is to provide balanced processing times among processes and avoid starvation. It is similar to round-robin scheduling, but with two major differences. First, CFS maintains process information in a red-black tree for constant time look up. Round-robin, on the other hand, uses a queue. Second, round-robin scheduler allocates a fixed execution period for each process while CFS uses a fair time period, with calculation based on several process parameters such as priority and percentage of CPU usage.

First-In-First-Out Scheduler. FIFO scheduler executes processes in the order that they are submitted to the run queue. The process keeps running until it finishes its work or it has to block as it awaits resources such as data being produced by other processes. In Linux, the FIFO scheduler exists on top of CFS. When an application is configured to use FIFO, threads within the application execute in a FIFO manner. However, threads in an application using FIFO are still subjected to preemption by the underlying CFS. To clarify the interplay between these two schedulers, we provide a simple example.

Assume that application A creates four threads: T_0, T_1, T_2, and T_3 in this particular order. If A is configured to use FIFO, its runtime behavior could be as follows:

- *Uni-processor*— T_0 is scheduled first. The other three remaining threads cannot run until T_0 finishes. For T_0, there is a maximum allowable time for it to run and if T_0 uses up this time, it is preempted so that *other threads from other applications can run*. The key point to take away from this example is that T_1 cannot run until T_0 is done. The execution order is from T_0 to T_3, respectively.
- *Multi-processor*— In Linux, there is a run queue per processor. Assume that our system has two processing cores so there are two run queues: R_0 and R_1. Even when FIFO is used, concurrent execution of two threads from the same application on two different cores is possible; e.g., T_0 is first placed on R_0 and T_1 is then placed on R_1. Note that T_2 and T_3 cannot be scheduled until T_0 or T_1 is done. The key point to take away from this FIFO policy example is that the number of concurrent threads from an application is determined by the numbers of cores and run queues in the system.

2.2 Garbage Collection

Currently, most state-of-the-art Java Virtual Machines use *Generational garbage collection* [13] to manage heap space. A generational collector segregates objects into "generations" using age as the main criterion. A typical generational garbage collector exploits the fact that objects have different lifetime characteristics; some objects are short-lived while others live for a long time. In terms of age distribution, past

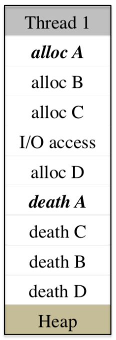


Figure 1: Calculation Lifespan of Object A

studies, mainly based on single threaded applications, have shown that “most objects die young” (referred to as the weak generational hypothesis [13]); and thus, collection of objects in the youngest generation (minor collection) occurs more frequently than collection of objects from the entire heaps (mature collection). Note that the parallel GC in OpenJDK 1.7 HotSpot, which is the platform used in this study, parallelizes the process of identifying reachable objects to improve performance [1].

In a minor collection, live objects are copied from the young generation to the next older generation [13], the amount of work that must be done by a typical minor collector is proportional to the amount of live objects that must be copied. This implies that a small number of surviving objects would result in less GC effort, a shorter collection pause, and higher *GC efficiency*, which refers to the collected memory as a fraction of the allocated space prior to a collection [15], [13].

The lifespan of an object can be computed by observing the number of objects that have been created or the amount of available memory in the young generation that has been used between the creation of that object and the time that it dies (i.e., the object is no longer reachable because there are no more references to it) [23], [22], [11], [13]. Figure 1 illustrates a case in which there are three objects (B, C, and D) created during the lifetime of object *A*. Therefore, the lifespan of *A* is the total allocated size of these three objects.

An important consideration is that *object allocations consume the available heap space, but operations on those objects (execution) progress them toward their deaths*. If these two factors are not balanced (e.g., a long allocation burst that fills up the heap without sufficient execution efforts), generational GC would be ineffective because each minor GC invocation would need to copy a large number of live objects, resulting in poor garbage collection performance. Next, we conduct experiments to observe the relationship between object lifespans and number of threads and how it can affect scalability.

3. Scalability of Parallel Garbage Collection

In this section, we evaluate the scalability of parallel GC in OpenJDK 1.7. We then investigate the relationship between the workload distribution among threads and the average object lifespan as we varied the number of threads and CPUs.

Benchmark	Suite	Version	Heap size(MB)	Scalable
avrora	DaCapo	2009	75	No
eclipse	DaCapo	2009	330	No
fop	Dacapo	2009	90	No
jython	DaCapo	2009	90	No
h2	DaCapo	2009	900	No
lusearch	DaCapo	2009	90	Yes
pmd	DaCapo	2009	210	No
sunflow	DaCapo	2009	210	Yes
tomcat	DaCapo	2009	135	No
xalan	DaCapo	2009	150	Yes
compiler.compiler	SPECJVM	2008	4000	Yes
compiler.sunflow	SPECJVM	2008	2500	Yes
compress	SPECJVM	2008	2500	Yes
crypto.aes	SPECJVM	2008	2500	Yes
crypto.rsa	SPECJVM	2008	2500	Yes
crypto.signverify	SPECJVM	2008	2500	Yes
derby	SPECJVM	2008	4000	Yes
scimark.fft.large	SPECJVM	2008	2500	No
scimark.lu.large	SPECJVM	2008	2500	No
scimark.sor.large	SPECJVM	2008	2500	No
scimark.sparse.large	SPECJVM	2008	2500	No
mpegaudio	SPECJVM	2008	2500	Yes
xml.transform	SPECJVM	2008	4000	Yes
xml.validation	SPECJVM	2008	4000	Yes

Table 1: Benchmarks used in this paper. The last column identifies whether the application is scalable based on our empirical study.

3.1 Experimental Methodology

We conducted our experiments on a NUMA machine with four AMD 6168 chips; each chip contains 12 processing cores and the total RAM is 64 GB. We used Ubuntu 14.04.2 LTS and the kernel version is 64-bit 3.16.0-30 as the operating system and OpenJDK 1.7 HotSpot as the Java Virtual Machine (JVM). We configured HotSpot to use the stop-the-world throughput-oriented parallel scavenge garbage collector. This collector is based on a two-generational scheme (young and mature), previously described in Section 2 [13], [1]. The collector utilizes multiple threads to scavenge the nursery generation. Objects that survive the young generation are copied to the mature generation. This particular collector, by default, uses as many threads as the number of available cores to perform nursery (young-generation) collection, therefore, in presenting results, we use *cores* to reflect this constraint.

We chose 24 benchmarks from Dacapo-9.12-Bach [4] and SPECJVM2008 [21] as listed in Table 1. The unselected benchmarks from both suits either cannot run on our platform (tradesoap and tradebeans [1]) or run too short because of small workloads (startup.* and *.small benchmarks). The workloads and number of driver threads (64) are fixed for all benchmarks with different settings, and the actual number of

[1] http://sourceforge.net/p/dacapobench/mailman/message/29144190/

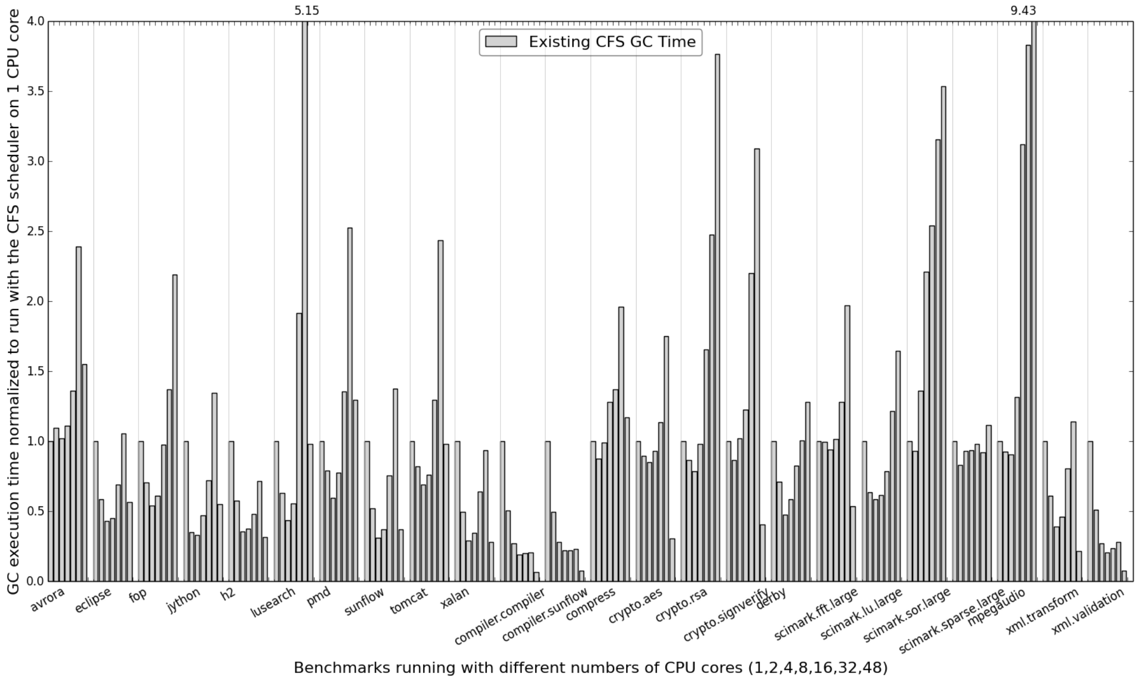


Figure 2: Effect of multiple threads/multiple cores on garbage collection scalability. For each application, there are seven bars. Each bar, from left to right, represents the GC execution time when 1, 2, 4, 8, 16, 32, and 48 threads and cores are used.

threads can be larger than the specified number of threads in each application.

Because the overall performances of Java applications can be sensitive to the workload and available heap space, we used the default input set and the same heap size for each run of an application (e.g., heap is set to 210MB for *sunflow* while the number of available CPU cores ranges from 1 to 48) as indicated in the Table 2. To set the heap size, we first identify the minimum heap requirement for an application using one mutator thread (i.e., if the heap is set to a smaller size, the application would crash). We then multiply this heap size by three (i.e., 3X of minimum heap size) [26]. Because there exists execution non-determinism due to threading, we ran each application 6 times to calculate the average value of each performance metric.

Table 2: Number of objects allocated (in million)

Benchmark	Number of mutator threads					
	2	4	8	16	32	48
eclipse	5308.5	5305.2	5306.8	5297.0	5304.7	5315.2
h2	3890.3	3897.6	3900.9	3906.4	3910.7	3914.1
jython	2217.8	2219.5	2217.7	2225.3	2225.6	2217.9
lusearch	5337.0	5336.5	5336.4	5336.4	5336.5	5336.9
pmd	*306.9*	*310.9*	*317.2*	*329.5*	*357.0*	*384.5*
sunflow	2529.3	2529.5	2529.9	2530.8	2532.7	2534.6
xalan	975.6	975.9	976.6	977.9	981.1	984.3

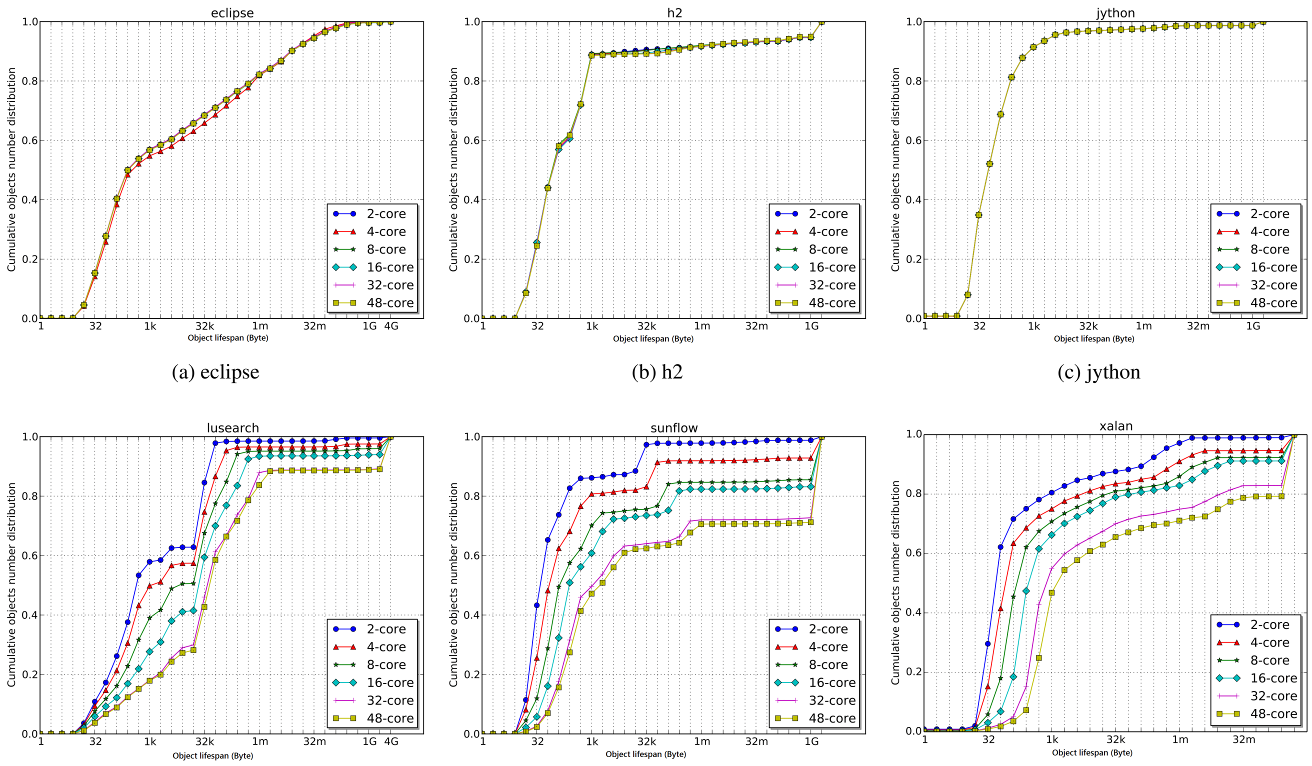
In JVM related studies, the experiments can be conducted with or without warm-up. Warm-up periods can provide time for the dynamic compilation system to generate optimized code. The approach with no warm-up periods has also been used in prior performance studies of application and garbage collection [5], [3]. In this study, we did not employ the warm-up approach in order to reduce the amount of traces. To validate the effects of using no warm-up periods, we verified our results with those reported by Du Bois et al. [6]. In that work, they attempt to distinguish scalable and non-scalable applications and evaluate garbage collection performance on multicore systems. Their methodology includes warm-up periods. We found our results and theirs to be nearly identical.

3.2 Scalability of Parallel GC

Previous studies have shown that parallel garbage collector can have poor scalability [6, 8–10, 17, 18]. As our first step, we conducted investigation to reconfirm the previously reported results for our setup. The results appear in Figure 2.

As shown, the GC times for most benchmarks are the smallest for four or eight threads/cores. Once we get to 16 threads/cores and beyond, we see that the GC times increase significantly for most applications. However, we also notice that increasing the number of threads in these application only slightly increases the number of allocated objects as shown in Table 2. As such, the additional garbage collection time is *not due to an increasing number of objects that must be collected.*

Based on this observation, we hypothesize that ineffective garbage collection, in which each collection invocation is not able to collect many objects, is the main reason for higher



(a) eclipse (b) h2 (c) jython

(d) lusearch (e) sunflow (f) xalan

Figure 3: Object lifespan characteristics of scalable and non-scalable applications. The x-axis represents the object lifespan in bytes. The y-axis indicates the percentage of total objects that have a particular lifespan.

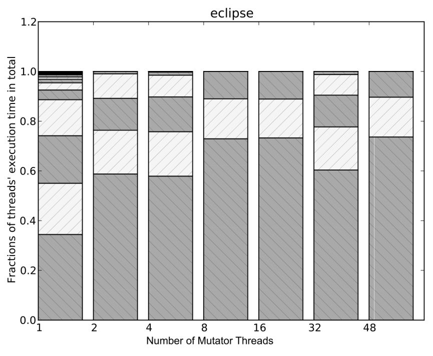
garbage collection overhead when 16 or more threads are used. Our hypothesis is based on the fact that generational collection performs well if most objects die young. However, a large number of surviving objects would results in higher copying cost [20], and therefore, higher garbage collection overhead. To validate our hypothesis, we conducted another experiment to evaluate the effects of increasing threads/cores on object lifespans.

Our results indicate that in some applications, object lifespans are sensitive to the numbers of threads and CPU cores while others show no sensitivity at all. Due to space limitation, we illustrate both characteristics in Figure 3 through six applications. The first three applications, *eclipse*, *h2*, and *jython*, show no change in object lifespans as we increase the number of threads. For the remaining three applications, *lusearch*, *sunflow*, and *xalan*, object lifespans are affected by increasing the number of the CPUs.

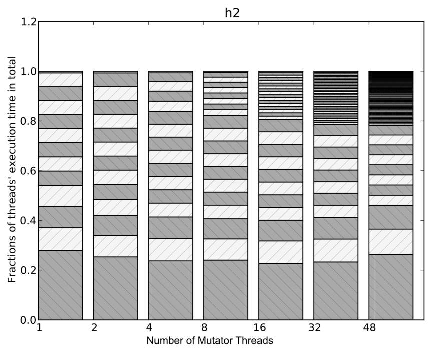
For *lusearch*, approximately 60% of all created objects have lifespans of 1KB or less when running with 2 mutator threads. When we set *lusearch* to run with 48 threads, less than 20% of all created objects have lifespans of 1KB or less. For *sunflow* and *xalan*, we observed a similar increase in lifespan. Over 80% of objects have the lifespans of 1KB or less when running with 4 mutator threads. However, only 50% of objects have the lifespans of 1KB or less when running with 48 threads. Such increased lifespans resulted in more objects surviving the nursery collection. Therefore, more time is spent copying these surviving objects to the mature generation and more full garbage collection invocations are needed as the mature region fills up more quickly.

Next we characterize these applications to better understand why their lifespan characteristics are so much different. As part of this investigation, we observe the workload distribution among threads. Using the same six benchmarks, we report the amount of work done by each thread in Figure 4. The first three applications, *eclipse*, *h2*, and *jython* employ only a small number of threads regardless of the number of available threads to help perform the work. For example, *jython* mainly uses three to four threads to do most of the work. In the case of *h2*, only four to twelve threads perform most of the work. As we increase the number of threads, these additional threads perform very little work. This uneven workload distribution greatly limits scalability because adding more threads does not reduce the amount of work that the main threads must do.

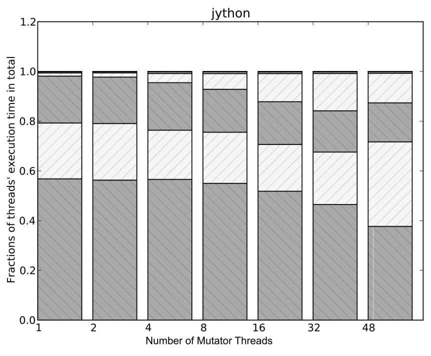
On the other hand, *lusearch*, *sunflow*, and *xalan* show nearly uniform distribution of workload among threads. As more threads are added during our experiment, each thread



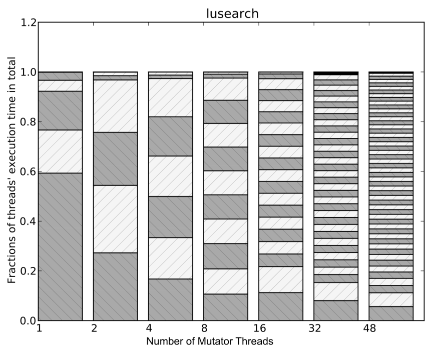
(a) eclipse



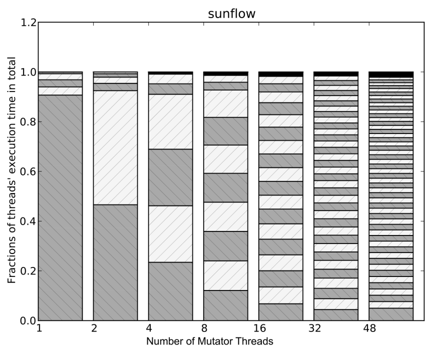
(b) h2



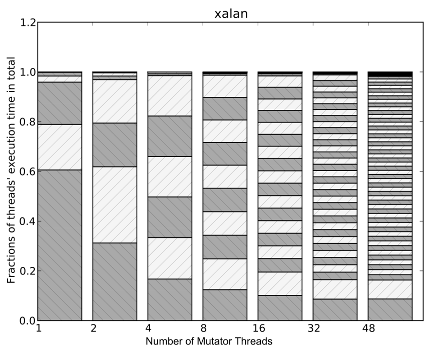
(c) jython



(d) lusearch



(e) sunflow



(f) xalan

Figure 4: Fractions of threads' execution time as the number of employed threads increases. We sort each bar by placing the thread that spends the most time on the bottom of the bar and the one with the least time at the top of the bar.

would perform proportionally less work. So, these three applications achieve the first objective of being scalable. The last column of Table 1 identifies whether an application is scalable based on the results of our experiment.

3.3 Discussion

When the heap is shared by multiple threads, thread scheduling performed by the underlying operating system can affect lifespans of objects belonging to a thread. In our example shown in Figure 1, a thread, ($T1$) allocates *object A*, *object B*, *object C*, and *object D*. Subsequently, $T1$ would access these objects but before doing so, it needs to perform an I/O operation. At this point, the operating system would suspend the execution of $T1$ and perform the requested I/O access on its behalf. If there are no other threads allocating objects from the same heap as $T1$, the lifespan of every object in *T1* can be easily calculated based on the object allocation pattern of $T1$. Thus thread scheduling by the operating system has no effect on lifespans in a single-threaded environment.

On the other hand, consider a scenario where three threads, $T1$, $T2$, and $T3$ share the same heap. When $T1$ is suspended by the operating system during the I/O access, the scheduler may pick $T2$ to run next and then subsequently $T3$. At this point, the calculation of the lifespan of *object A* must include objects created by $T2$ and $T3$. While the execution of $T1$ does not depend on any objects created by $T2$ and $T3$, these objects can greatly prolong the lifespans of objects created by $T1$. Furthermore, they also create additional allocation pressure that would eventually result in a minor collection invocation.

For the three scalable applications, evenly distributed workload means that the heap is shared by many threads, causing these applications to suffer from significantly prolonged lifespans due to scheduling and more threads allocating/using objects concurrently. This can result in poorer GC effectiveness as more time is spent on moving surviving objects from one generation to the next. For the remaining applications, additional threads perform little work and therefore, they have little effects on lifespans. In the next section, we describe our approach to reduce the effect of scheduling on object lifespan.

4. Effects of FIFO vs. CFS

In this section, we compare GC scalabilities and overall performances of the two scheduling policies: FIFO and CFS using the 24 benchmarks.

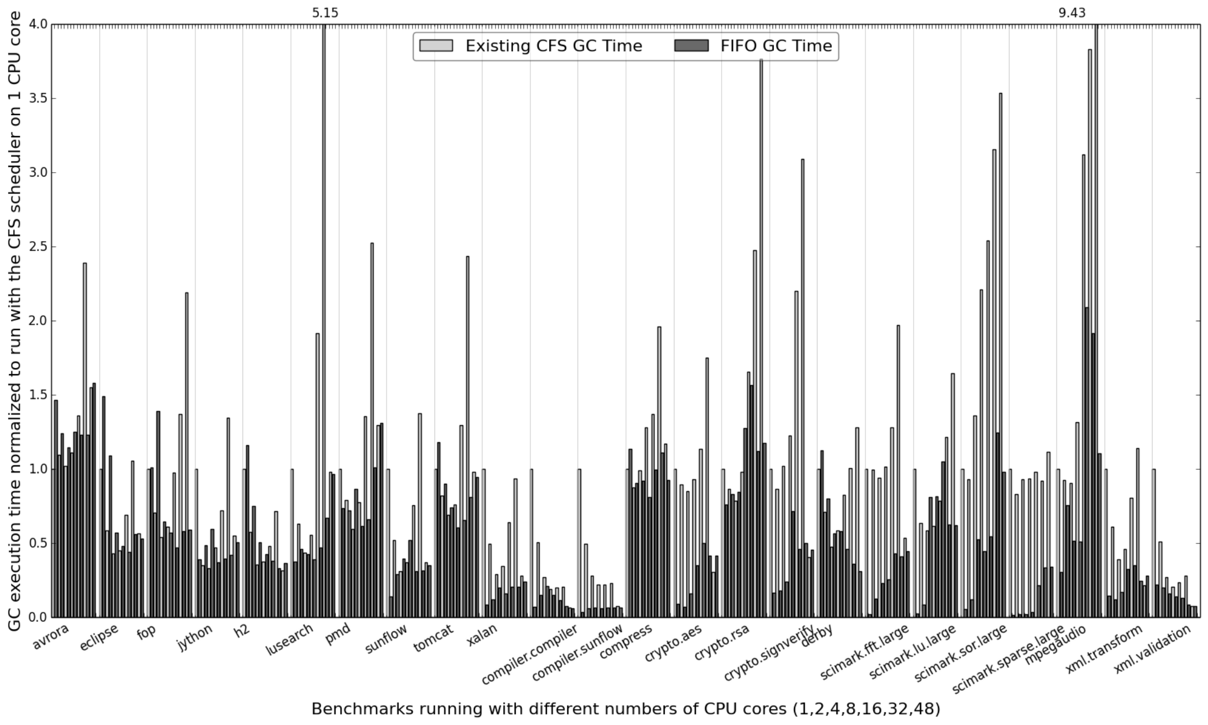


Figure 5: GC scalability with the CFS and FIFO schedulers. Each application has 14 bars. The gray bars represent CFS and the black bars represent FIFO. From left to right for each application, the pairs of gray and black bars indicate GC execution times for the number of cores = 1, 2, 4, 8, 16, 32, and 48, respectively.

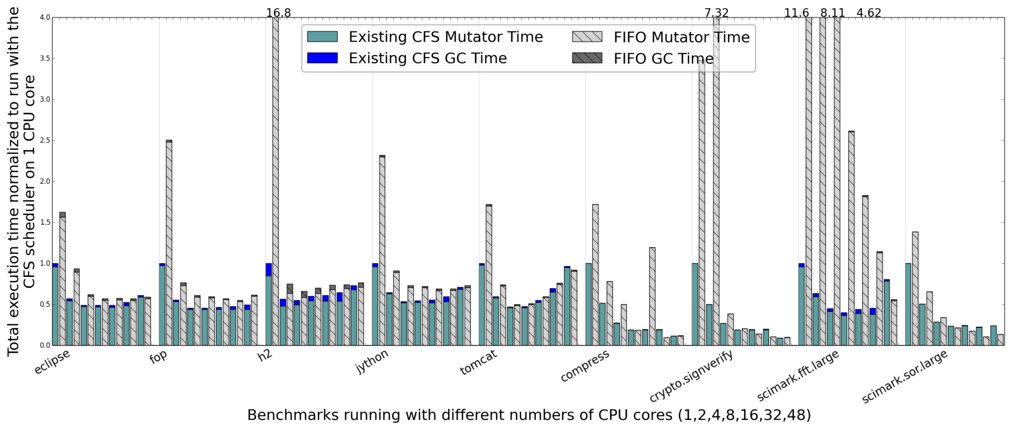


Figure 6: Comparison of the total execution times for CFS and FIFO schedulers for applications that do not benefit from FIFO. The bars for each application, from left to right, are for runs with 1, 2, 4, 8, 16, 32, and 48 cores.

4.1 GC Scalability Study

Figure 5 reports the CG scalabilities of each application when CFS and FIFO are used. The GC times of each application using CFS and FIFO are normalized to that of the same application using CFS running on a single CPU. We observe that the GC execution times when FIFO is used are smaller than those when CFS is used. The average GC execution time is reduced by 52%, while the biggest GC time reduction is nearly 90% for *scimark.fft.large*. Further, although GC performance for FIFO can degrade for higher numbers of threads/cores, it is still better than that for CFS. We find that the reason for the better performance is shorter object lifespans in FIFO due to less heap competition. FIFO also has fewer context switches during the GC period.

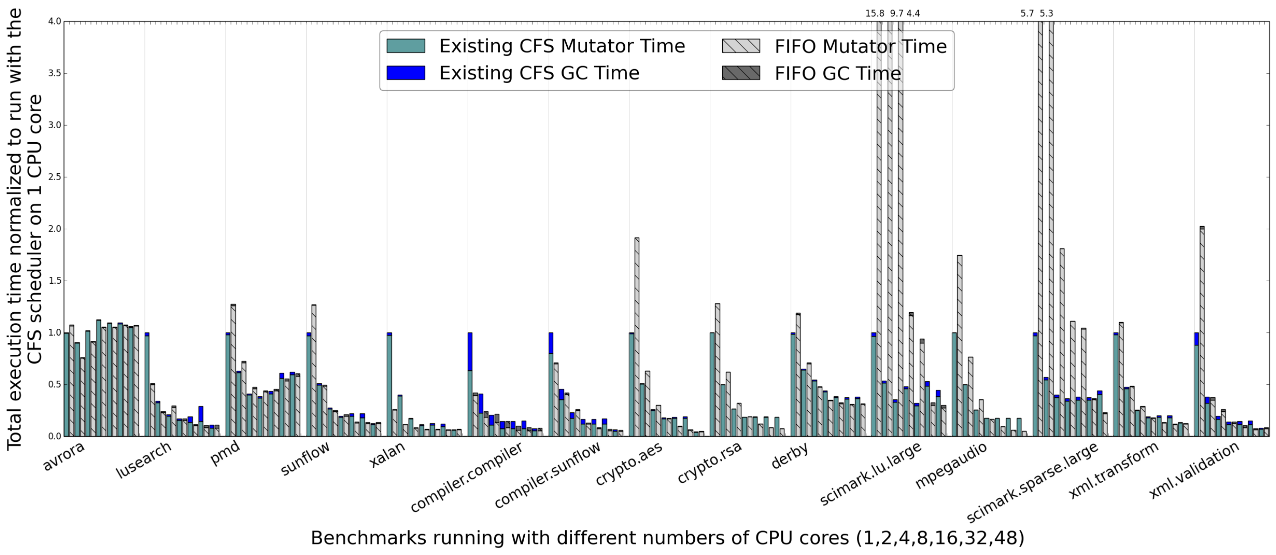


Figure 7: Comparison of the overall performances when CFS and FIFO are used. This figure reports the performances of applications that benefit from FIFO. We configured applications and system to use 1, 2, 4, 8, 16, 32, and 48 cores.

4.2 Study of Mutator and GC Performances

Although FIFO improves the GC performance over CFS with more CPUs, some mutator times become longer in comparison as shown in Figure 6. We note, however, that the benchmarks that work better with the CFS scheduler are all non-scalable, as previously characterized in Figure 4. Most of the threads in these benchmarks are helper threads and perform very little work and can finish within the CFS quanta. As such, the FIFO scheduler does not have any advantage over the CFS scheduler for these benchmarks. This is also true for *avrora* and *pmd*, which have been identified as non-scalable. However, in these two applications, FIFO and CFS performances are very similar, as shown in Figure 7.

Figure 7 presents the scalable benchmarks that show performance improvement when FIFO is used. The average improvement of the overall performance is 45% with 74% as the maximum improvement in *crypto.aes*. We also observed context switching behaviors of these scalable applications when FIFO is used. On average, FIFO reduces context switching by 90% over CFS. Fewer context switches result in better cache locality for threads. On the other hand, when only a few threads in an application do most of the work, more context switches occur when helper threads need to run and therefore they preempt the major threads [16].

4.3 Effects of FIFO on Other Applications Using CFS

By using two scheduling algorithms in a system, it is possible that a policy used in one application can affect schedulability of other applications, and therefore, affect their overall performance. To evaluate the impacts of FIFO on the performance of applications configured to use CFS, we conduct a set of experiments. First, we measure (i) performance of an application when FIFO is used, (ii) performance of an application when CFS is used, (iii) performances of two instances of the same application both running FIFO, (iv) performances of two instances of the same application, one using FIFO and the other using CFS, (v) performances of two instances of the same application, both running CFS. We set these programs to use 48 threads/cores. For brevity, we report the results of a scalable (*sunflow*) and non-scalable (*eclipse*) applications.

In Figure 8 (top), *sunflow*, which is a scalable application, performs best when FIFO is used. When we run two instances using FIFO, we see a slight increase in execution times. This is expected as there are more threads in the system. When we run two instances with two different schedulers, we also see a slight increase in execution time for FIFO but a more significant increase for CFS (about 27% higher than that of running alone with CFS). When both instances use CFS, all 96 or more threads from these instances can run in any order without having to wait for others to finish. This intensifies the competition to create objects in the heap, resulting higher GC overhead and execution time. When FIFO is paired up with CFS, the competition becomes less intense so the performance of CFS does not degrade as much as when both instances run CFS.

In Figure 8 (bottom), *eclipse*, which is a non-scalable application, shows that the impact of the scheduler change is very little when two instances run concurrently. One reason is that FIFO yields no benefit in non-scalable applications so the performance difference between FIFO and CFS is small. Another reason is that few threads in the non-scalable applications do most of the work; thus, there are idle CPUs that other applications can use. Fewer working threads also means less heap competition among threads within the application.

4.4 Discussion

In nearly all benchmarks (scalable or non-scalable), FIFO can provide benefit if used in a system with a large number of cores. It tends to perform worst when the number of cores is fewer than eight. This is to be expected as FIFO can achieve parallelism with more cores. In scenarios with a few processing cores, threads that are scheduled earlier would prevent other threads from running.

While FIFO can work well in most applications when more than 8 cores are used, there are also benchmarks that perform better with CFS. In addition, large server applications that rely on working threads polling for work would also perform poorly with FIFO. Up to now, we need to manually identify whether an application can benefit from using FIFO. The selection criteria include scalability and clearly defined amount of work. Next we introduce two approaches to automatically detect applications that can benefit from using FIFO. These mechanisms are only enabled when more than 8 processing cores are used. Otherwise, the system continues to use the CFS policy.

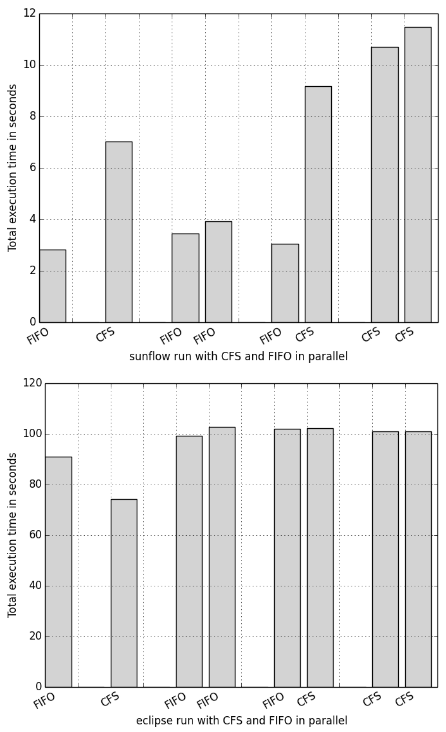


Figure 8: Comparison of the overall performances when CFS and FIFO scheduled same applications run in concurrent on 48 CPUs

5. Automatic Selection of Scheduling Policies

The analysis and results presented in the previous section indicate that the workload distribution between threads can help select a more suitable scheduling policy. In this section, we present two approaches to automatically select an optimal scheduling policy for an application.

5.1 Using Cross-run Profiles

The first approach transparently collects workload distribution information in the background of one execution run (profiling run) and uses it to select a better scheduling policy for the subsequent runs. Algorithm 1 describes the steps used in the approach. We measure the execution time of each thread as the workload. The collection of the threads' information is done by reading the system file */proc/$pid/stat*, which requires one thread pinned to a CPU to monitor the file. Figure 9 compares the performance degradation when the profiling script is executed concurrently with the application. As shown, in the worst case, the profiling thread adds 8% to the execution time. Keep in mind that this is a one time cost as it only incurs in the profiling run.

Algorithm 1: Proposed sampling based algorithm

```
1 Initialization: launch the multi-threaded Java
  application (J) and the helper script (H) concurrently;
2 while J still runs do
3     collect the active threads;
4     record current timestamp;
5 end
6 calculate the execution time of each thread;
```

Figure 10 demonstrates the three types of profiles, indicating whether CFS or FIFO should be selected or both schedulers should have similar performances. If most work is done by few threads (*jython*, the first bar in the figure), CFS performs better because few involuntary context switches are needed as explained in the previous sections. Otherwise, if the workload is well distributed among most or all threads (*lusearch*, the last bar in the figure), FIFO performs better as it improves the memory locality. For some applications, the workload is distributed between some threads (*avrora*, the middle bar), the performance difference between using CFS and FIFO policies is small.

5.2 Adaptive Runtime Selection

Because many factors such as underlying architecture, heap size configuration, and phase changes in application behaviors, it would be more efficient to adaptively select a better performing scheduler at the runtime, instead of employing a profiling run. In this section, we present a low-overhead sampling based scheduler selection mechanism that can adaptively switch between the CFS and FIFO schedulers according to the execution profiles of an application.

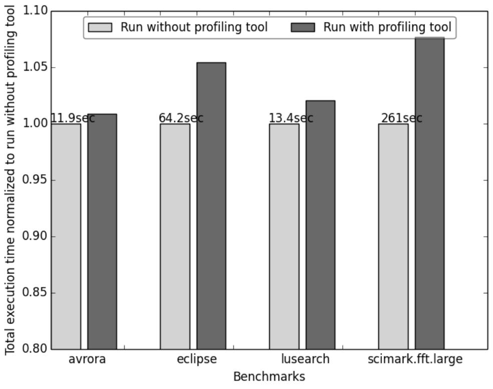


Figure 9: Overhead of the Profiling Thread.

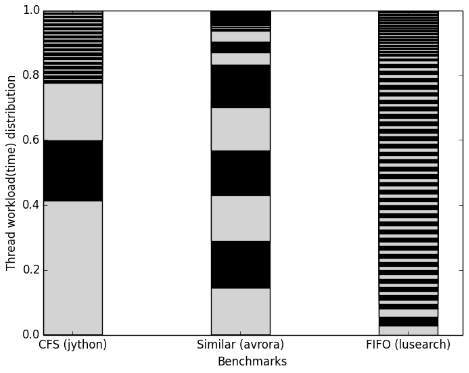


Figure 10: Three Different Workload Distribution Patterns.

Algorithm 2 describes the steps using in the proposed mechanism. The design exploits our observation that non-scalable applications tend to employ a small number of threads to do most of the work. As such, these threads are often long running. On the other hand, scalable applications tend to use threads more uniformly, and therefore, they often have similar execution times. In our approach, we start the profiling thread runs along side the application threads. The tool collects Thread IDs (TIDs) of running and suspended threads every n seconds. Currently, n is predefined as a fixed number (i.e., 2 seconds) for all the benchmarks. The selection process is explained later in this section. However, we can speculate that dynamically changing n for each application can provide more accurate profiles. We leave this part as future work.

Our mechanism compares the TIDs of two successive collections. If the TIDs are mostly the same, the mechanism selects CFS as the scheduler. If the TIDs are different, the mechanism selects FIFO scheduler. With this sampling approach, our system can deal with phase changes in an application. For example, an application, by default, starts with CFS. After a few sampling, the mechanism may switch it to FIFO. Toward the end when most threads have died and only a few threads remain, it can switch back to CFS. While it is possible that the same application can start a thread, let it die, and then start another thread that has the same TID as the first thread, it is very unlikely. For that to happen, 2^{22} threads must be created in the system and the same application is assigned the same TID twice.

The sampling thread incurs very low-overhead because it often performs sampling only a few times during an application execution and the amount of work required to read the system file names in the process tree provided by the Linux is quite small. The comparison of TIDs between two successive sampling is done with the hash-table container which is done in constant time. The small number of TIDs also means that the amount of memory required to store these TIDs is small.

In addition to using the TIDs, we also explore the following strategies: monitoring the number of context switches and execution time distribution among threads. We found the performance results to be either similar to the proposed approach of using TID, as in the case of using context switching, or worse due to the high overhead (e.g., I/O and computation) or sampling noises that can degrade the selection accuracy, as in the case if using execution time distribution among threads.

Algorithm 2: Proposed adaptive runtime selection algorithm

```
1  Initialization: launch the multi-threaded Java
   application (J) and the helper script (H) concurrently;
2  J starts with the FIFO scheduler;
3  while J still runs do
4      collect the active threads into a hash-table;
5      if one thread is already in the hash-table then
6          Switch the scheduler to the CFS;
7          break;
8      else
9          Switch the scheduler to the FIFO;
10     end
11     wait for period time n;
12 end
```

The decision of value n. In this work, n is selected as 2 seconds. But we experimented with other values ranging from 1 second to 16 seconds as well which shows 2 seconds is the best number for all benchmarks. The value of n is determined quasi-dynamically, which needs the tuning in the executions. As stated previously, the future improvement is to adaptively change the scheduler based on dynamic value of n.

5.2.1 Evaluation

The proposed adaptive mechanism is used to adaptively select policies. We then compare its performance with the ap-

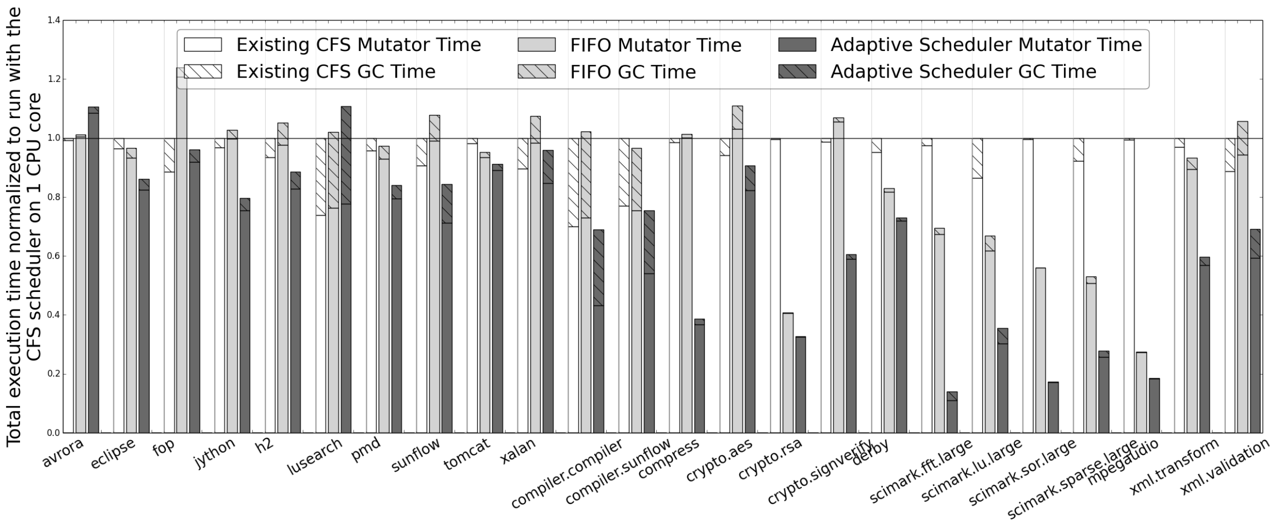


Figure 11: Evaluation of the proposed sampling algorithm.

proach of manually setting the policy to be CFS of FIFO. Figure 11 reports the results of our investigation.

Except for *eclipse* and *avrora*, the proposed adaptive policy selection mechanism performs better than the manually selected CFS and FIFO schedulers. We see that the adaptive mechanism can remain in CFS based on the detection of long running threads, implying that most work is done by those threads, in non-scalable applications. We also see that when an application uses many helper threads to prepare work for the main threads, there is a phase in this application that can benefit from FIFO. However, once this phase has passed, only a few worker threads remain and the system should switch back to CFS. Our mechanism is able to make the initial switch from CFS to FIFO, and then the subsequent switch from FIFO back to CFS.

6. Related Work

This section discusses some of the prior related work on performance improvement of multi-threaded Java applications from various aspects.

Design of scheduling algorithms for Java application. Xian et. al [24–27] design several types of schedulers for Java applications, including object allocation phase aware and contention aware schedulers. The allocation phase aware scheduler schedules the threads according to their allocation rate or preempt threads with the memory consumption instead of the running time which is used in the CFS. The contention aware scheduler avoids the threads with possible shared data running at the same time to reduce the lock contention, which improves the parallelism and so the execution time. Sartor et. al [19] investigate the performance difference when mapping the JVM threads and Java applications' threads to different and same CPU sockets on NUMA. The insights in their work include there is a cost for mapping, remote memory traffic degrades the performance and pinning threads to different sockets does not always improve the performance.

Performance limited by resources and synchronizations. Main sources of contention include architecture resources such as processor cores and memory bus. Chen et al. [5] study the scalability of Java applications on different numbers of processor cores and threads. They measure hardware performance and use the findings to explain the scalability issues. The solution they proposed is thread-local allocation buffer that is for application instead of VM. Unlike our work, their focus is mainly on the application performance and not the combined performances of application and GC, and so is Huang's work [12]. Prior studies and our work show that garbage collection and heap usage can also negatively impact scalability. Kalibera et al. [14] study the non-scalable issue of DaCapo benchmarks and they focus on concurrency issues by measuring the memory usage, synchronization, and communication among threads. Their work provides a good insight into thread management in JVM but pay very little attention to the GC subsystem as a potential factor that limits scalability.

Performance of mutator and garbage collection in Java. Esmaeilzadeh et al. [7] report that not all multi-threaded Java applications are running shorter with more threads. Their work focuses on the performance improvement with respect to architecture such as increased processing cores, instead of program's characteristics that can limit scalability. Gidra et al. [9, 10] compare the scalability of various garbage collection algorithms in JVM and report that time spent on stop-the-world GC increases with number of GC threads. This is the same observation as ours. However, they attribute this overhead to heap synchronization and objects movement on NUMA. Du Bois et al. [6] use Bottle Graphs to characterize work done by JVM and application threads when different numbers of threads/cores are used. They conclude that GC needs to do more work when there are more mutator threads. However, they do not further explore why more GC work is needed. Our work investigates the reasons for this observation in much greater details. We believe that both application and JVM need to be scalable in order to achieve the highest possible scalability.

7. Conclusion

Execution parallelism can be a double-edged sword. It can improve execution performance by having more threads collaborate on performing the work, but the collaboration makes threads compete for heap resource and degrade GC performance. Such contentions become aggravated with the CFS scheduler in the Linux. Without changing either the application, the JVM or the system, our investigation shows that many Java applications perform better with FIFO than with CFS. The improvement is gained through higher GC efficiency and mutator performance.

Based on this observation, we propose two approaches to optimally select scheduling policies based on application scalability profile. Our first approach uses the profile information from one execution to tune the subsequent executions. Our second approach dynamically collects profile information and performs policy selection during execution. Our evaluation result indicates that the dynamic selection process works well in our test subjects and can effectively switch scheduling policies based on dynamic application behaviors.

Acknowledgment

This material is based on research sponsored by NSF, DARPA and Maryland Procurement Office under agreement numbers CNS-1116606, FA8750-14-2-0053 and H98230-14-C-0140, respectively Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. Government.

References

[1] Memory Management in the Java HotSpot Virtual Machine. `http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`, 2006.

[2] Completely Fair Scheduler in Linux. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`, 2015.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and Realities: The performance Impact of Garbage Collection. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):25–36, 2004.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[5] K.-Y. Chen, J. M. Chang, and T.-W. Hou. Multithreading in Java: Performance and Scalability on Multicore Systems. *Computers, IEEE Transactions on*, 60(11):1521–1534, 2011.

[6] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications. In *ACM SIGPLAN Notices*, volume 48, pages 355–372. ACM, 2013.

[7] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley. Looking Back on the Language and Hardware Revolutions: Measured Power, Performance, and Scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 319–332. ACM, 2011.

[8] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the Scalability of Garbage Collectors on Many Cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, page 7. ACM, 2011.

[9] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, et al. A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores. In *ASPLOS 13-Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 229–240, 2013.

[10] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. NumaGiC: a Garbage Collector for Big Data on Big NUMA Machines. In *ASPLOS 15-Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, 2015.

[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] W. Huang, J. Lin, Z. Zhang, and J. M. Chang. Performance Characterization of Java Applications on SMT Processors. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 102–111. IEEE, 2005.

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

[14] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A Black-box Approach to Understanding Concurrency in DaCapo. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 335–354, 2012.

[15] C.-T. D. Lo, W. Srisa-an, and J. M. Chang. A Quantitative Simulator for Dynamic Memory Managers. In *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on*, pages 64–69. IEEE, 2000.

[16] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson. *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.

[17] J. Qian, D. Li, W. Srisa-an, H. Jiang, and S. Seth. Factors Affecting Scalability of Multithreaded Java Applications on

Manycore System. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 167–168. IEEE, 2015.

[18] J. Qian, W. Srisa-an, D. Li, H. Jiang, and S. Seth. SmartStealing: Analysis and Optimization of Work Stealing in Parallel Garbage Collection for Java VM. In *Proceedings of the 12th International Conference on Principles and Practice of Programming in Java*, pages 170–181. ACM, 2015.

[19] J. B. Sartor and L. Eeckhout. Exploring Multi-threaded Java Application Performance on Multicore Hardware. In *ACM SIGPLAN Notices*, volume 47, pages 281–296. ACM, 2012.

[20] R. Shahriyar, S. M. Blackburn, and K. S. McKinley. Fast Conservative Garbage Collection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 121–139. ACM, 2014.

[21] J. SPEC. Client2008 Suite. *URL: https://www. spec.org/ jvm2008/*, 2008.

[22] D. Stefanović, M. Hertz, S. M. Blackburn, K. S. McKinley, and J. E. B. Moss. Older-First Garbage Collection in Practice: Evaluation in a Java Virtual Machine. *SIGPLAN Notices*, 38(2 supplement):25–36, 2003.

[23] 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.

[24] F. Xian, W. Srisa-an, C. Jia, and H. Jiang. AS-GC: An Efficient Generational Garbage Gollector for Java Application Servers. In *ECOOP 2007–Object-Oriented Programming*, pages 126–150. Springer, 2007.

[25] F. Xian, W. Srisa-an, and H. Jiang. Allocation-phase Aware Thread Scheduling Policies to Improve Garbage Collection Performance. In *Proceedings of the 6th international symposium on Memory management*, pages 79–90. ACM, 2007.

[26] F. Xian, W. Srisa-an, and H. Jiang. Microphase: An Approach to Proactively Invoking Garbage Collection for Improved Performance. In *ACM SIGPLAN Notices*, volume 42, pages 77–96. ACM, 2007.

[27] F. Xian, W. Srisa-an, and H. Jiang. Contention-aware Scheduler: Unlocking Execution Parallelism in Multithreaded Java Programs. In *ACM Sigplan Notices*, volume 43, pages 163–180. ACM, 2008.