SmartStealing: Analysis and Optimization of Work Stealing in Parallel Garbage Collection for Java VM

Junjie Qian
University of Nebraska Lincoln
Lincoln, Nebraska USA
jqian@cse.unl.edu

Witawas Srisa-an
University of Nebraska Lincoln
Lincoln, Nebraska USA
witty@cse.unl.edu

Du Li
Carnegie Mellon University
Pittsburgh, PA USA
duli@cs.cmu.edu

Hong Jiang
University of Nebraska Lincoln
Lincoln, Nebraska USA
jiang@cse.unl.edu

Sharad Seth
University of Nebraska Lincoln
Lincoln, Nebraska USA
seth@cse.unl.edu

Yaodong Yang
University of Nebraska Lincoln
Lincoln, Nebraska USA
yyang@cse.unl.edu

ABSTRACT

Parallel garbage collection has been used to speedup the collection process on multicore architectures. Similar to other parallel techniques, balancing the workload among threads is critical to ensuring good overall collection performance. To this end, work stealing is employed by the current state-of-the-art Java Virtual Machine, OpenJDK, to keep GC threads from idling during a collection process. However, we found that the current algorithm is not efficient. Its usage can often cause GC performance to be worse than when work stealing is not used.

In this paper, we identify three factors that affect work stealing efficiency: determining tasks that can benefit from stealing, frequency with which to attempt stealing, and performance impacts of failed stealing attempts. Based on this analysis, we propose SmartStealing, a new algorithm that can automatically decide whether to attempt stealing at a particular point during execution. If stealing is attempted, it can efficiently identify a task to steal from. We then compare the collection performances when (i) the default work stealing algorithm is used, (ii) work stealing is not used at all, and (iii) the *SmartStealing* approach is used. Without modifying the remaining garbage collection system, the evaluation result shows that *SmartStealing* can reduce the parallel GC execution time for 19 of the 21 benchmarks. The average reduction is 50.4% and the highest reduction is 78.7%. We also investigate the performances of SmartStealing on NUMA and UMA architectures.

CCS Concepts

•Software and its engineering → Memory management; Garbage collection; Multithreading;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPPJ '15, September 08-11, 2015, Melbourne, FL, USA

© 2015 ACM. ISBN 978-1-4503-3712-0/15/09…$15.00

DOI: http://dx.doi.org/10.1145/2807426.2807441

Keywords

Parallel garbage collection; Work stealing

1. INTRODUCTION

Java bytecode is an instruction set executed by Java Virtual Machine (JVM). Initially, Java was the only language that compiled its programs into Java bytecodes. However, the benefits of utilizing JVM including portability and ease of memory management through garbage collection, have inspired language designers to create additional languages such as Scala and Clojure [24, 16] that target the JVM. In addition, JVM support for multithreading also makes these languages attractive for the development of applications that can exploit thread-level parallelism available in modern multi-core processors.

Similar to application programs, garbage collectors in JVMs often exploit thread-level parallelism to improve collection performance on multicore architectures. Both stop-the-world and concurrent collectors have a parallel phase which enables multiple GC threads to do the collection concurrently, and to avoid contention when a GC thread fetches a GC task to mark the reachable objects in parallel [15, 11, 27, 2]. As such, the performance of a parallel collector is determined by the longest running thread, which is typically a thread that has been assigned the most amount of GC work (e.g., traversing the biggest or most complex object graph).

Good scalability of a parallel collector means that the GC execution time would be shortened with more threads and CPU cores [8]. Most current studies on improving the scalability of parallel GC have been done on the NUMA many core architecture, addressing the remote memory access problem and lock free queue implementations [29, 11, 12, 28]. As the parallel phase accounts for most of the GC time, the imbalanced workload among GC threads also limits the performance and scalability of a parallel collector.

Work stealing [20, 14] has been adopted as a load balancing technique in parallel garbage collection [2, 13]. In work stealing, when there are no more root tasks available in the task queue, an idle GC thread is assigned to a stealing task, which tries to find a currently running GC thread and steal one of its tasks to work on. The key to its effectiveness in parallel GC lies in deciding when to steal and which object to steal, otherwise, the collection would be unnecessarily prolonged by futile stealing attempts when there is no work

or little work to steal from.

The previous studies on work stealing in GC have focused on new data structures that eliminate lock contention and on improved algorithms that better balance the load between threads doing stealing and normal tasks [9, 19, 21]. On the other hand, our study tries to find factors that can impact work stealing efficiency. In our study we compare the performances of a parallel collector with work stealing to the same collector with no work stealing. The results show that the parallel collector with the default work stealing performs worse than the one without work stealing in many cases. In the cases where work stealing fares better, they tend to be the ones in which the numbers of stealing tasks are very small.

We analyze the experimental results and identify factors that negatively impact the efficiency of the work stealing as: (1) blind stealing even when there is little or no work to steal from; (2) a naive and indiscriminate selection of GC tasks from which to steal; and (3) frequent but futile attempts to steal. Based on these observations, we implement a low overhead runtime work stealing algorithm, called SmartStealing, which can decide whether to steal in the first place and, if so, which currently running GC thread to steal from to improve the overall parallel GC performance. The proposed algorithm is an add-on to the current default work stealing mechanism to optimize its decision. As such, it can also be applied to work stealing algorithms in other parallel collectors.

We evaluate the performance of the proposed SmartStealing on both the UMA and NUMA systems. The results reveal mixed impacts on both work stealing algorithms (the default one and SmartStealing). That is, work stealing, in general, works well in our UMA system but performs poorly in our NUMA system. We think that the main reason for such poor performance is that work stealing has been designed for UMA systems and is not aware of remote memory access in NUMA systems. For future work, we plan to develop efficient work stealing algorithm for NUMA systems.

In summary, this paper makes the following contributions:

- Identification of the factors affecting the efficiency of work stealing. Further, the profiling results are presented to show the relationships between these factors and the stealing performance.
- Comparisons of the GC performances of parallel GC with and without work stealing on different architecture platforms and different application environments. The conclusion is that neither scheme works for all environments. Rather, judiciously selecting a small number of GC stealing tasks works best. This suggests that an algorithm is needed to automatically select the right number of stealing tasks and to steal at the right time.
- The design and implementation of SmartStealing, a low- overhead work stealing mechanism that helps decide when and how to do the stealing at runtime based on the information collected at runtime.

The reminder of the paper is organized as follows. Section 2 introduces the background on the parallel GC and work stealing in OpenJDK's parallel GC. Section 3 explains the platform, tools and implementations we used in this paper. Section 4 demonstrates the experiment results that motivate our research. Section 5 presents the comparisons of different work stealing implementations and illustrates the design of *SmartStealing* Section 6 evaluates the proposed algorithm.

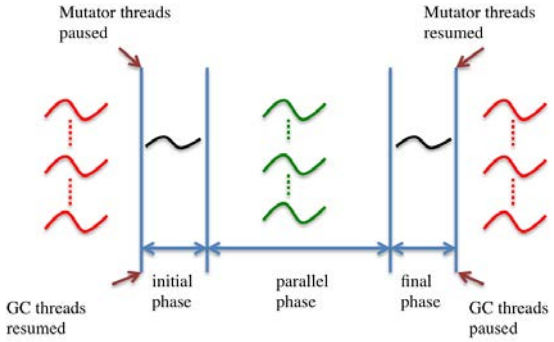


Figure 1: The phases of Parallel GC

Section 7 discusses the related work. Section 8 concludes our work.

2. BACKGROUND AND RELATED WORK

This section provides the background information on the parallel garbage collector used in Oracle OpenJDK7 and on its three GC phases.

2.1 Parallel Collector

The parallel collection algorithm employed by Oracle OpenJDK7 and used in this study is a stop-the-world type, which pauses application execution before performing any collection work. The application resumes its execution after collection finishes. The algorithm uses multiple GC threads to perform parallel copying in the young generation and parallel mark-compact in the tenured generation [2]. To understand how this particular type of collector behaves, we investigate its collection phases and its mechanism to ensure load balancing through "work stealing". We also investigate the effects of recently introduced optimization approaches on GC performance. Ultimately, we focus our evaluation on its efficiency and scalability.

2.2 GC Phases

Both young and tenured generation parallel collectors divide the collection efforts into three phases, initial, parallel, and final, as shown in Figure 1.

Initial phase. When a heap region is full or GC is manually invoked by the application, the VM stops mutator threads using barrier synchronization, prepares the tasks queue, and invokes GC threads to perform parallel collection. The tasks queue is a lock protected queue, consisting of various kinds of tasks which mostly are thread root task, stealing task and barrier task. The initial phase also identifies the root sets, which mostly are system classes, JNI locals/ globals, threads and native stacks [1]. These root sets can be accessed by different tasks.

Parallel phase. All the GC threads are released to acquire one task each from the task queue. They serially fetch tasks from the lock-protected queue. To fetch a task, all the GC threads compete to acquire the lock and then fetch one task from the top of the queue. Once a thread successfully fetches a task, it can begin execution. All the threads execute in parallel. Although the GC tasks may vary in the work they do, the most crucial work is marking reachable objects in the heap by traversing the heap starting at root sets. Because some tasks may take longer to finish, load balancing is

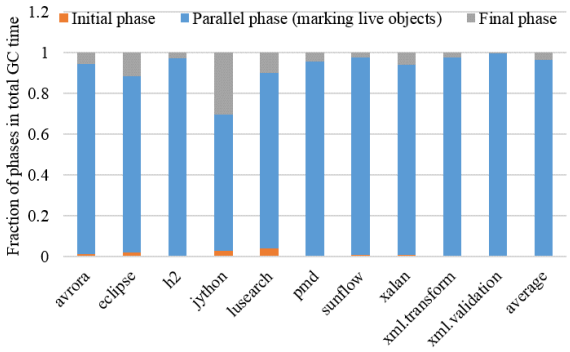


Figure 2: The time distribution of the three GC phases on UMA

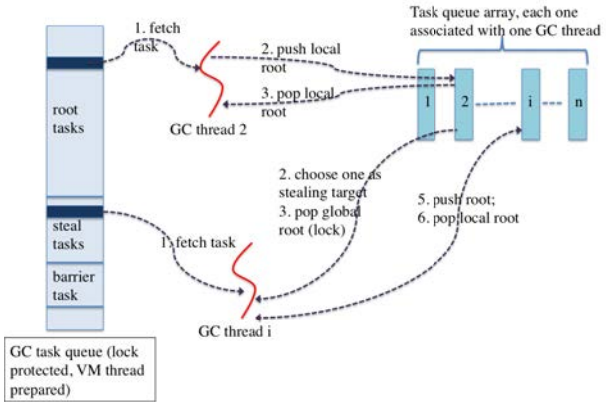


Figure 3: The work flow of Parallel GC threads fetching and executing tasks

performed to evenly distribute the remaining tasks through *work staling*. The basic idea is to steal a task assigned to a busy GC thread and assign it to an idle thread. When a GC thread is assigned a barrier task, which is typically the last task in the work queue, the GC thread checks that all other GC threads have stopped before calling a VM thread to enter the final phase.

Final phase. There is only one virtual machine thread working in this phase. During a minor collection, the objects marked as live by the GC threads would be copied to the old or tenured generation. During a mature collection, live objects are moved as part of the compaction process. Additional finalization works is also performed in this phase. This includes checking all GC threads have stopped, re-sizing the heap size if necessary, and resuming the mutator threads. When the final phase finishes, the mutator threads resume execution and the GC threads are blocked until next garbage collection or application termination, whichever occurs first.

Compared with the initial and final phases, the parallel phase consistently accounts for the longest time in each garbage collection invocation as shown in Figure 2, which demonstrates the fraction of different phases' time in the total GC time with a single GC thread on the UMA architecture. Because the other two phases employ single-threaded execution, the improvement of parallel phase can determine the overall improvement in garbage collection efficiency in multicore architectures. Furthermore, the scalability of parallel GC also depends on the scalability of the multiple GC threads working in the parallel phase.

2.3 Work Stealing in Parallel GC

Figure 3 highlights the basic steps in parallel GC. For effective utilization of multiple GC threads in the parallel phase, it is important to ensure even distribution of work to these GC threads. There are several factors that can cause GC workload to be unbalanced among threads. First, a single lock protecting the task queue can become a major bottleneck and cause delays in assigning tasks to GC threads. In this scenario, many threads would remain idle as they wait for tasks. Second, the complexity of the assigned tasks can vary substantially for various reasons. A GC thread visits all the reachable objects on heap from its assigned root and the number of reachable objects from the assigned roots can vary greatly. Third, some GC threads may scan for intergenerational references to determine whether the object is live or not. Such scan operation to uncover the hidden references in the heap can prolong the GC thread's collection work.

Work stealing is a mechanism that attempts to balance the GC workload in parallel GC [9]. It is done by assigning the stealing tasks to the idle GC threads. The number of the stealing tasks is equal to the number of the parallel GC threads. As such, it is possible for a thread to attempt stealing even when there is nothing to steal. Such futile attempts can add overhead to the overall GC time.

Algorithm 1 explains the existing work stealing. When there are GC tasks available for stealing, the decision as to which task to steal from can be based on the depth of that task's local stack, used as the proxy for the amount of its pending work. To simplify the selection process, two local stacks are randomly picked from the running threads and the one with the longer stack depth is chosen for stealing. However, this selection process has been identified as one possible GC overhead [10].

Algorithm 1: The existing work-stealing algorithm

```
1  Initialization: a GC thread (T) gets a stealing task;
2  Steal_begins:
3  randomly select two running GC threads;
4  pick the thread with deeper stack;
5  pop the bottom of the selected thread's stack;
6  if fails to pop then
7  |   goto Steal_begins;
8  else
9  |   T finishes the marking task;
10 |   goto Steal_begins;
11 end
12 GC terminates the GC threads;
13 Steal_ends:
```

3. METHODOLOGY

In this section, we provide information about the experimental platforms that we used to conduct our studies. The software platform includes the host OS, JVM, the benchmarks, and the configurations used to run these benchmarks. Note that knowing the JVM configuration is crucial for re-

producibility of results, as JVM performance including that of HotSpot is sensitive to settings such as the heap size and heap layout (e.g., ratio between young and old generations). We also specify the system hardware, including the number of CPUs, the memory architecture, and the amount of main memory.

3.1 Software Platform

Benchmarks. We choose the benchmarks from DaCapo-9.12 [3] and SPECJVM2008 [25] as shown in Table 1. The applications from DaCapo suite are selected, except *tradesoap, tradebeans* because they cannot run in our experimental platforms. All the benchmarks are multithreaded, and some from Dacapo are internal multithreaded although they are single thread driven. The heap sizes used to run the benchmarks are also included in Table 1.

These benchmarks are commonly used by researchers to conduct memory management and Java systems research. They have varying heap requirements, support different number of threads, and provide different garbage collection workload. Some benchmarks require large heap sizes and result in long garbage collection times. Small heap requirement often means short collection time. Such heap variability also creates variable load balancing requirements.

We run each benchmark three times to attenuate possible noises, because the execution can be affected by other processes in the system and the scheduling mechanism. Each time, a new instance of the JVM is created (instead of running an application three times in a single instance of the JVM). We did not have any warm-up runs, because our focus is on load balancing in parallel garbage collection and not the optimizing compiler in the JVM.

Benchmark	Suite	Version	Heap size(MB)
avrora	DaCapo	2009	75
eclipse	DaCapo	2009	330
jython	DaCapo	2009	90
h2	DaCapo	2009	900
lusearch	DaCapo	2009	90
pmd	DaCapo	2009	210
sunflow	DaCapo	2009	210
tomcat	DaCapo	2009	135
xalan	DaCapo	2009	150
xml.transform	SPEC	2008	4000
xml.validation	SPEC	2008	4000
compiler.compiler	SPEC	2008	4000
compiler.sunflow	SPEC	2008	2500
compress	SPEC	2008	2500
crypto.aes	SPEC	2008	2500
crypto.rsa	SPEC	2008	2500
crypto.signverify	SPEC	2008	2500
mpegaudio	SPEC	2008	2500
scimark.sparse.small	SPEC	2008	2500
scimark.fft.small	SPEC	2008	2500
scimark.lu.small	SPEC	2008	2500

Table 1: Benchmarks used in this paper

Java virtual machine. Our studies were conducted on OpenJDK7, which is a widely used commercial open-source JVM from Oracle. OpenJDK7 provides many useful flags to report the GC information such as GC thread execution time stamps and the time stamps of each GC task. The garbage collection algorithms provided in OpenJDK7 include Parallel Scavenge, G1 and Concurrent Mark-Sweep. In this work, we used Parallel Scavenge, which is a stop-the-world parallel garbage collector as described in Section 2.

Operating system. The operating system we used is Ubuntu 14.04.2 LTS and the kernel version is 64-bit 3.16.0-30. The difference on operating system affects the build of Java virtual machine, and the underlying memory management of the virtual pages.

3.2 Hardware platforms

We use both NUMA and UMA architectures in this work. The NUMA architecture has four processors; each with 12 cores. As such the system has 48 processing cores. It also has 64GB of main memory (16GB per processor). The *-XX:+UseNUMA* flag is enabled for each run on NUMA platform. The UMA is single processor with 8 cores and 16GB of main memory. The detailed information of both platforms are shown in Table 2.

Architecture	NUMA	UMA
Model	AMD Opteron 6168	Intel Xeon X3440
Number of sockets	4	1
Cores per socket	12	8
Memory	64 GB	16 GB

Table 2: Hardware platform characteristics

4. MOTIVATION

We first investigate factors that can impact GC performance and scalability. The results reveal various performance and scalability issues that are discussed below.

4.1 Imbalanced GC Workload

Figure 4 reports the execution time of each GC thread on the randomly selected *Lusearch* benchmark from the DaCapo suite [3], running on a NUMA machine. In the figure, there are nine GC threads and only four of the GC threads get the root tasks. Three of the nine GC threads remain idle as they try to access the task queue. Although each GC thread is also assigned a stealing task, three have nothing to steal.

Note that at the beginning of the parallel phase, GC threads are not assigned tasks at the same time due to contention on the lock that protects the task queue. As such, some GC threads idle for a long period of time. One way to eliminate these delays is to adopt lock free queue [11]. However, synchronization overhead still exists among GC threads.

At the end of the parallel phase, we also see that some threads are idling as they have already finished their GC tasks while other threads have not. These idling threads are then assigned stealing tasks to help out other GC threads. Towards the end of the parallel phase, the number of threads assigned the stealing tasks exceeds the number of busy GC threads. As a result, several stealing tasks look for tasks but fail to steal. In many cases, the actual GC task is already done before the conclusions of these futile attempts, causing additional delays before the GC can finish.

4.2 Selecting Stealing Targets

There are three types of stealing methods, `steal_1_random`, `steal_best_of_all` and `steal_best_of_2`. The default one

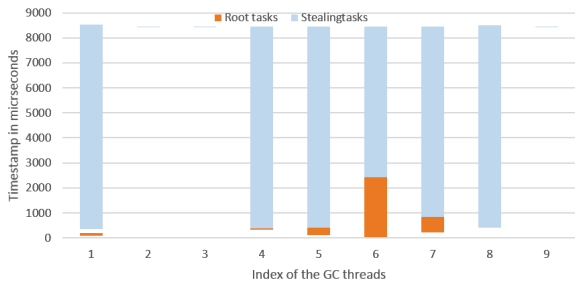


Figure 4: Load imbalance of Parallel GC

is `steal_best_of_2`, which randomly picks out 2 of all currently running GC threads and target on the one which has the deepest stack. We investigate the cost of selecting a stealing target from the two choices.

We used *Oprofile* [17] to determine the number of instructions needed to perform `steal_best_of_2` and compute the fraction of these instructions over the entire set of GC instructions. The experimental results show that in UMA architecture an average of 5% and the maximum of 11.6% (for avrora) of the total GC time is spent in determining stealing targets. A similar result was also previously reported by Gidra et al. [10]. There are two main reasons for this overhead: First, the selection process occurs frequently; Table 3) shows that each GC task makes hundreds of stealing attempts to determine the stealing target. Second, the process to make the stealing choice is not implemented in an efficient way. It randomly selects two potential targets and compares their stack depths. However, the stack depths can change during this process, as other GC threads continue to add unvisited fields to or pop objects from the stack while other GC threads steal from the stack. This results in the cache lines of the parameters (stack depths of two chosen threads) having to be updated frequently, which is overhead. Meanwhile, because the stack depths may change after the comparison but before the stealing thread gets the object, the comparison result may expire and the stealing thread does not steal from the new longer one, which makes the function to select target meaningless.

	NUMA	UMA
avrora	69	173
eclipse	132	644
h2	628	4730
jython	41	42
lusearch	52	11
pmd	208	599
sunflow	137	109
tomcat	84	87
xalan	72	97
xml.transform	288	366
xml.validation	1942	1816

Table 3: Average number of stealing attempts by a single stealing task

4.3 Futile Stealing Attempts

A large number of stealing attempts per task can impact performance in both positive and negative ways. The positive impact is that attempts to perform load balancing would be timely. However, stealing attempts at the wrong time (e.g., toward the end of the parallel phase) can result in an unnecessary overhead resulting from excessive selection processes and unsuccessful attempts at stealing. Table 4 shows the fraction of the successful stealing attempts over the total stealing attempts.

	NUMA	UMA
avrora	0.52	0.84
eclipse	0.64	0.85
h2	0.68	0.66
jython	0.18	0.69
lusearch	0.35	0.37
pmd	0.83	0.98
sunflow	0.75	0.87
tomcat	0.6	0.9
xalan	0.53	0.9
xml.transform	0.87	0.9
xml.validation	0.98	0.98
average	0.64	0.86

Table 4: Fraction of successful steals over all attempts

As shown, some benchmarks have high percentages of successful stealing. However, on the average, 20% of the attempts fail (total failures in total attempts for all benchmarks) and contribute to possibly avoidable stealing overhead. Also note that the average success rates are higher in UMA than those of NUMA because our UMA system has fewer processing cores than the NUMA system. As such, fewer GC threads are used in the UMA systems compared with those in the NUMA system. Because the amount GC work is approximately the same between the two systems (i.e., we use the same number of application threads), having more stealing tasks would result in lower success rate.

The experimental results also show that once a stealing task fails to steal, it tends to fail again in its subsequent attempts (shown in Table 5). We conjecture this is because most of the failures happen at the end of the collection work which means there is little or no collection work to be stolen. Also note that the failure rates after the first failure in the UMA system are lower than those in the NUMA system. Again, this is due to a smaller number of the GC threads in the UMA system.

	NUMA	UMA
avrora	0.67 (16)	0.72 (369)
eclipse	0.68 (11303)	0.3 (26057)
h2	0.52 (11963)	0.39 (52848)
jython	0.76 (674)	0.31 (2243)
lusearch	0.99 (872)	0.45 (167)
pmd	0.87 (131)	0.1 (70)
sunflow	0.90 (170)	0.38 (564)
tomcat	0.96 (97)	0.42 (49)
xalan	0.99 (486)	0.34 (365)
xml.transform	0.87 (2258)	0.41 (12623)
xml.validation	0.83 (1168)	0.17 (533)

Table 5: Fraction of subsequent failures after the first failure over all failures (the number of failures after the first failure)

4.4 Communication Overhead in NUMA

In a NUMA system, a remote memory access is costlier than a local memory access [4]. As the work stealing process in parallel GC is not cognizant of the NUMA architecture, the GC threads attempt to steal from a remote node, incurring high memory copying costs between the two nodes [11]. Table 6 shows that most of the steals occur on targets residing on remote nodes.

	remote	local
avrora	95741 (61.9%)	58950 (38.1%)
eclipse	2144814 (72.1%)	831513 (27.9%)
h2	874104 (74.0%)	307276 (26.0%)
jython	1010443 (76.4%)	311579 (23.6%)
lusearch	1931795 (87.0%)	289576 (13.0%)
pmd	192758 (81.4%)	44066 (18.6%)
sunflow	345666 (85.8%)	57423 (14.2%)
tomcat	309400 (84.0%)	58940 (16.0%)
xalan	1177270 (88.3%)	156318 (11.7%)
xml.transform	1443457 (76.2%)	451889 (23.8%)
xml.validation	1624142 (82.4%)	346768 (17.6%)

Table 6: Number of stealing to remote and local nodes

These issues, identified with the current implementation of work stealing, motivated us to modify the existing algorithm to address the issues. We call the modified algorithm SmartStealing, and discuss it next.

5. SMARTSTEALING ALGORITHM

Prior to introducing the proposed SmartStealing technique, we conducted a preliminary investigation to identify opportunities to optimize the current work stealing algorithm used in HotSpot. We accomplished this by comparing the performance of the parallel GC using the default work stealing mechanism with that of a parallel GC configured using no work stealing. This is done to evaluate the benefits of work stealing. We report the result of our preliminary investigation next.

5.1 Optimization Opportunities

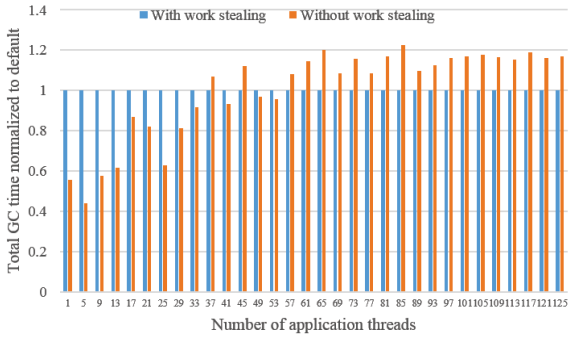
To identify issues and potential bottlenecks in the current implementation, we need to monitor the effects of application threads, GC threads, and stealing tasks on GC performance. Our experiments vary these configurations as we enabled and disabled work stealing mechanism on our experimental UMA and NUMA platforms (detailed information about our platforms is given in Section 3). The results are reported in Figure 5.

Also note that for brevity, we only illustrate these optimization opportunities through two applications, *pmd* and *xml.transform*. These two applications show two distinct performance characteristics. Specifically, *pmd* shows that work stealing is very beneficial in the UMA system while *xml.transform* shows that in about half of the configurations, we are better off not using work stealing. They also show very different performance profiles when running on the NUMA system.

Changing the number of application threads also affects the numbers of the available root tasks for the GC threads in each GC period. This is because each thread has its own runtime stack, using more threads would result in having



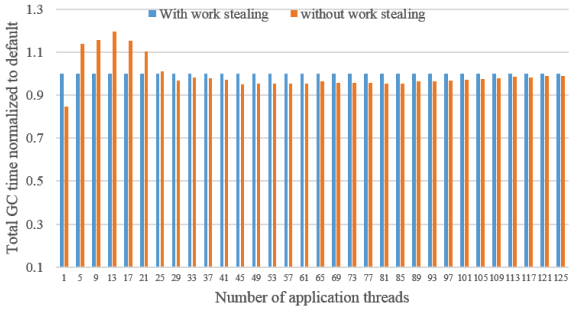
(a) pmd on NUMA



(b) xml.transform on NUMA



(c) pmd on UMA



(d) xml.transform on UMA

Figure 5: Comparison of different application threads' GC performances with and without work stealing on NUMA and UMA architectures

more runtime stacks. Each stack is a major source of root tasks in a program. In addition, having more application threads can also affect heap requirement and organization due to additional memory pressure. These factors can affect the performance of the work stealing mechanism. We also find that the performance of work stealing is sensitive to the hardware platform.

We also experimented with changing the number of GC threads. This change also affects the number of stealing tasks as it is currently set to be the same number as that of the GC threads. Work by [10], [23] and [7] have shown that 2-4 GC threads tend to yield the best GC performance, the experiments here are designed to show the impacts of different numbers of the GC threads on the efficiency of work stealing. Each benchmark is run with the same setting on the same platform except for the different numbers of parallel GC threads (again, due to the difference in the numbers of processing cores in our two platforms). The number of parallel GC threads is set to be fewer than the number of processing cores on the system to avoid resource contentions between the GC threads.

The results in Figure 6 show that applications behave differently with different numbers of the GC threads. On the NUMA platform, *pmd* performs better with no work stealing. On the other hand, *xml.transform* performs better with work stealing in many configurations. In general, work stealing does not seem to help GC performance in the NUMA system but seems to work slightly better in the UMA system. The exception is in *xml.transform* where work staling shows no benefits (except with 4 GC threads) in the UMA system and some benefit in the NUMA system in some configurations.

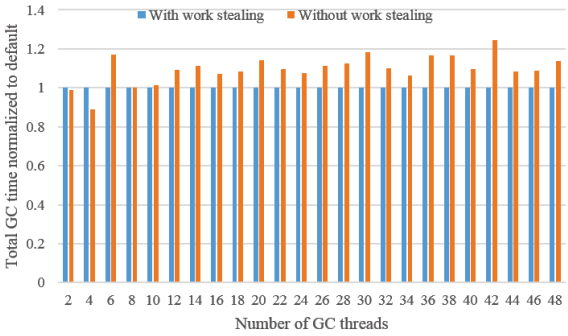
Next, we experimented with controlling the number of stealing tasks. In this experiment, we set the number of stealing tasks to be fewer than the number of GC threads. The number of the GC threads is fixed, and only first few stealing tasks are executed. Figure 7 reports the experimental results. The plot includes the best performance that can be obtained by varying the number of stealing task. For example, when we run *xml.transform* with 8 threads, the configuration that yields the best GC performance is when the number of stealing task is set to 2. However, as we increase the number to application threads to 32, the optimal number of stealing tasks becomes 6. The take away is that there is not an optimal number of stealing tasks that works best for all configurations of an application. In general, the number of stealing tasks should be smaller than or equal to the number of GC threads.

In summary, we observe that setting the number of stealing tasks to be the same as the number of GC threads does not yield the best performance. We also found that in some cases, work stealing does not yield any benefit. This is because these applications suffer from many futile stealing attempts that result in repetitive but useless invocations of the stealing target selection process. As such, disabling work stealing in these applications yields the best GC performance. We also find that work stealing is more effective in the UMA system than the NUMA system.

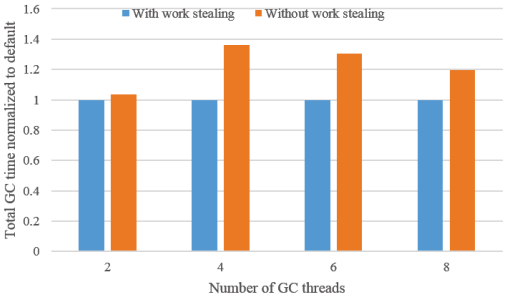
In designing SmartStealing, our two major objectives are to dynamically vary the number of stealing tasks to maintain the balance with the remaining GC tasks and be able to turn off work stealing when the application can no longer benefit from it. Next, we discuss the proposed



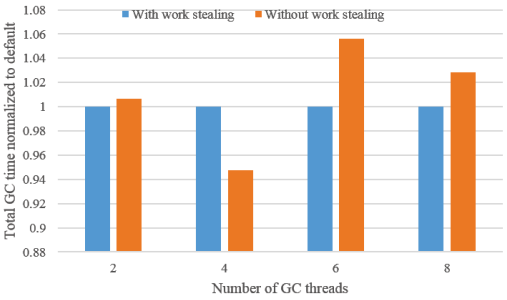
(a) pmd on NUMA



(b) xml.transform on NUMA



(c) pmd on UMA



(d) xml.transform on UMA

Figure 6: Comparison of different GC threads' GC performances with and without work stealing on UMA architecture

SmartStealing algorithm.

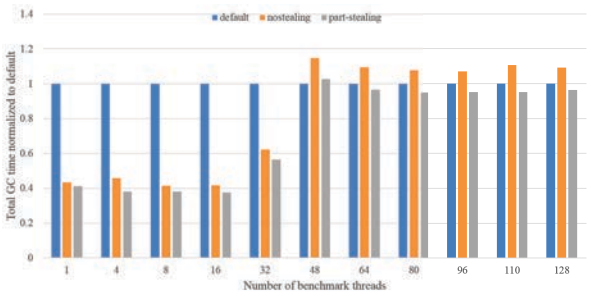


Figure 7: Comparison of fewer stealing tasks, default always stealing and no stealing at all

5.2 Design of SmartStealing

We propose an improved work stealing algorithm called SmartStealing that has been designed to deal with existing issues in the current work stealing algorithm by disabling stealing when there are no achievable benefits, making the target selection process more efficient, and adjusting the number of staling tasks to maintain a good balance with the amount of remaining GC work. We then evaluate its performance in Section 6 by comparing its performance with that of the default work stealing and that of no stealing at all. [1]

Two key insights employed by SmartStealing to improve the existing work stealing algorithm are: (1) gradually reducing the number of stealing tasks to maintain a good balance with the amount of remaining GC work (line 9 in Algorithm 2) and (2) reducing the frequency to select stealing targets (line 2-4 in Algorithm 2). The algorithm of the proposed SmartStealing is described in Algorithm 2. This is a component of the parallel phase in the GC, and is invoked when the stealing tasks are assigned to GC threads. Recalling that in the parallel phase, parallel GC threads get root tasks before attempting to perform work stealing. There is also a barrier task for execution after the work stealing.

When a GC thread attempts to steal a task, it first needs to get the information from `gcTaskManager` to determine whether there are other running tasks. If there are none, it skips the execution of this stealing task and tries to get the next available task or is terminated by the VM if there are no more tasks available for execution. The stealing choice is randomly made in order to eliminate the negative impact of `steal_best_of_2`. On the other hand, randomly selecting one task guarantees that there is always one stealing target.

When a thread assigned to a stealing task finishes its work, it would try to find another task to perform in an attempt to help other busy GC threads. The current work stealing implementation randomly selects two stacks and determines which is deeper. It then takes a GC task from the bottom of that stack and starts working on it. There are several problems with the current strategy. First, it needs to perform another selection process, which introduces more cost. Second, the selection process may also fail if the number of GC tasks is nearly or has completely exhausted.

Our design overcomes these two limitations by forcing the stealing task to fetch another GC task from the same stack until all GC tasks on that stack have been exhausted. After that happens, it then perform the target selection process again. This optimization eliminates the need to keep computing the next target. If it fails to find a new target, it would suspend itself to gradually reduce the number of stealing tasks in the system. This is done to maintain a good balance between the number of GC tasks and stealing tasks.

[1]The implementation of the proposed *SmartStealing* in OpenJDK7 is available on https://github.com/junjieqian/smartstealing.

Algorithm 2: The algorithm of *SmartStealing*

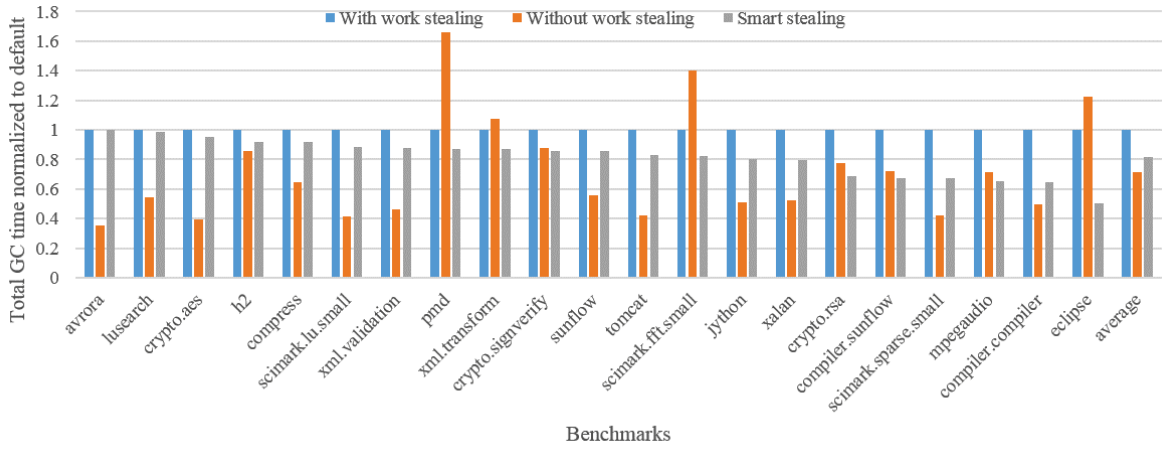
```
1  Initialization: a GC thread (T) gets a stealing task;
2  if there is no other task running then
3  |  T skips this stealing task;
4  |  T continues to the next available task;
5  else
6  |  Steal_begins:
7  |  randomly select a running GC thread;
8  |  pop the bottom of the selected thread's stack;
9  |  force termination if failed to steal;
10 |  T finishes the marking task;
11 |  while T is not terminated by the VM do
12 |  |  if the depth of last selected thread's stack is
   |  |  larger than 1 then
13 |  |  |  pop the bottom of the target thread's stack;
14 |  |  |  force termination if failed to steal;
15 |  |  |  T finishes the marking task;
16 |  |  else
17 |  |  |  goto Steal_begins;
18 |  |  end
19 |  end
20 end
```

6. EXPERIMENTAL RESULTS

The proposed SmartStealing is evaluated on both NUMA and UMA architectures, compared with default (always stealing) and no stealing at all implementations. All the benchmarks are run with m application threads and n GC threads, where m is the number of available processors (48 for NUMA and 8 for UMA) and n is 32 for NUMA and 8 for UMA architecture which are default settings in OpenJDK JVM. We use large working sets for the Dacapo benchmarks and 100 ops for the SPECJVM benchmarks.

Figure 8 presents the total GC time (normalized to the default, always stealing, implementation) for *SmartStealing*, default and no work stealing at all. Overall, the following observations can be made. First, no work stealing performs better in most cases than the default always work stealing design. Second, our new SmartStealing implementation improves the performance significantly. Compared with the current always stealing strategy, the new design can shorten the total GC time by up to 50% and nearly all benchmarks are beneficial from the new design. Next we analyze the results in more details:

1. The no stealing implementation is better than the SmartStealing implementation in some benchmarks, because although the SmartStealing avoids unneeded stealing tries, it still does stealing to balance the work-



(a) Comparisons on NUMA



(b) Comparisons on UMA

Figure 8: Evaluation of GC performances with smart stealing

loads. When the available work for stealing consists of small object reference graphs, the cost to do the stealing can exceed the gain from a balanced workload. The performance tuning to decide whether the work is worth the while to steal is important, and we leave this as future work.

2. The improvements of the SmartStealing algorithm for all the benchmarks on NUMA are similar as the improvements of the algorithm on UMA. This can be attributed to the benchmarks' features, such as the complexity of the heap graphs and the root tasks (e.g. the application threads in the stack).
3. On average, the SmartStealing algorithm is not better than no stealing on NUMA but is better than no stealing on UMA. This is due to the remote memory copying on NUMA. The stealing GC thread may steal from the GC threads running on the remote nodes, and so has to copy data from a remote memory bank, which results in overhead. On the other hand, no stealing cannot help with load balance but avoids remote memory copying overhead.

We also evaluate the SmartStealing algorithm with same benchmarks but different application threads. The SmartStealing algorithm works for most of cases but for the two benchmarks on UMA as shown in Figure 9. For these two exceptional cases, the default work stealing is always the best, which means that the load between the GC threads is imbalanced and without the work stealing most of the GC time may be spent by one or few GC threads. The SmartStealing algorithm stops stealing if a failure happens, which limits the load sharing ability and results in poor load balance performance.

We also investigated the scalability of the parallel GC in the UMA and NUMA systems. Figure 10 compares the total GC execution times for the default, no stealing and smart stealing approaches using two benchmarks that demonstrate comparable performances with the rest of the benchmarks. In the experiment, we varied the numbers of GC threads. As shown in the figure, both work stealing approaches (the default and SmartStealing) perform well in the UMA system. Therefore, it is not surprising to see that they generally result in better GC scalability than that of the GC with no



(a) avrora



(b) h2

Figure 9: The benchmarks that do not work with the smart stealing

stealing. However, we also observed that work stealing generally degrades the scalability of GC in the NUMA system. We think that the main reason for such poor performance and scalability is that work stealing has been designed for UMA and is not aware of remote memory access. New designs are needed for the latest NUMA architectures to improve the overall performance of parallel GC. We plan to address this challenge as our future work.

7. RELATED WORK

In response to the aforementioned issues in work stealing, Oancea et al. [21] allow each GC thread to trace only its own work list, which is associated with the local heap . This design improves the locality of the GC threads to the different heap regions, but does not help with the load imbalance. Iyengar et al. [15] propose a non-blocking and lock-free scheme for each GC thread to mark the objects, and a new work-sharing algorithm between the GC threads to better balance the workload. This design focuses on solving the contentions for marking bits between the GC threads and on dividing the marking work between the GC threads, instead of improving the work stealing mechanism.

Ideally, the collection time of a parallel collector should decrease proportionally with the addition of more GC threads on more CPU cores. However, several prior research efforts [10, 11, 7, 12, 23, 5] have indicated that the parallel collector in OpenJDK does not scale with more GC threads and more CPU cores. In fact, its GC performance degrades significantly when the number of GC threads is more than 4. There are many reasons contributing to the non-scalability of the parallel GC. These factors, identified in previous studies, include architectural changes that improve execution performance but degrade GC performance and increasing number of mutator threads.

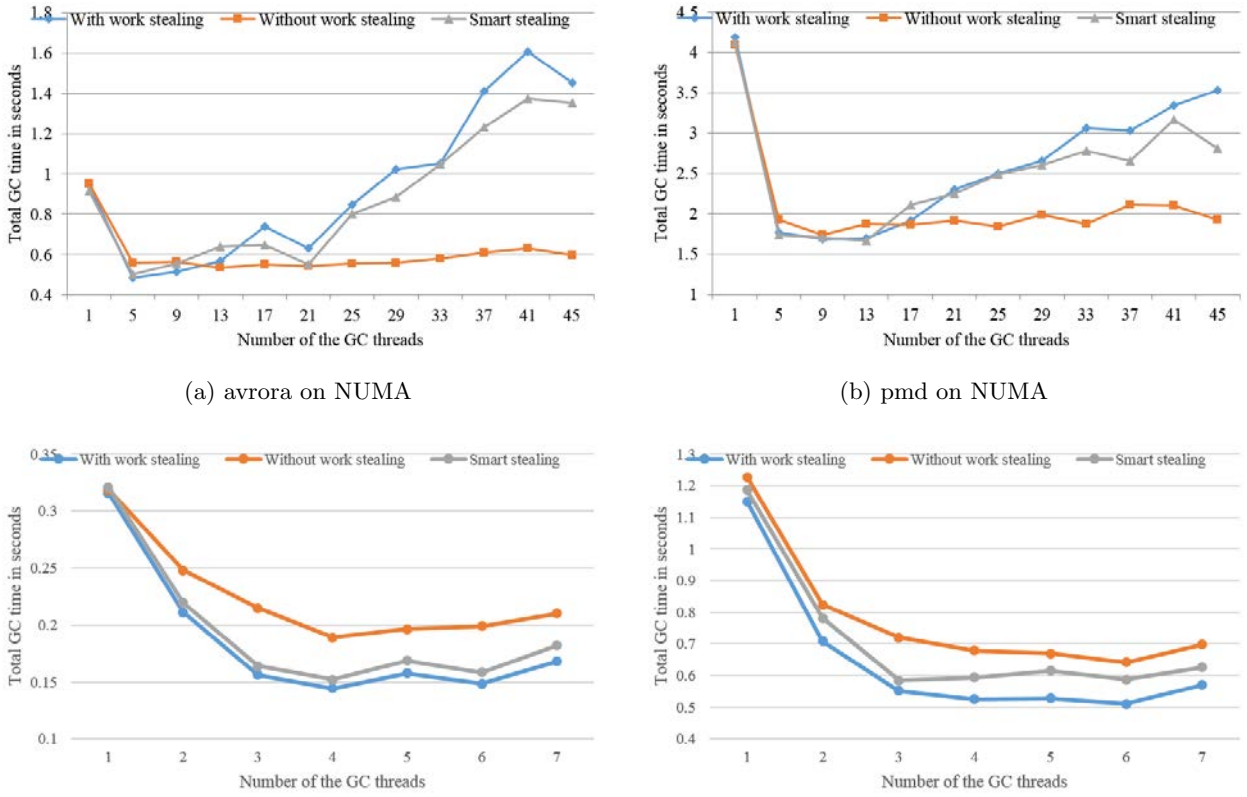
Gidra et al. [10, 11, 12] explain the impact of remote memory accesses in NUMA architecture on GC performance. They point out that when parallel GC threads have to mark the objects located on remote nodes, the communication overhead can be quite large and increase further when more GC threads and CPU cores are involved. This is one factor that limits the scalability of parallel GC. Gidra et al. then propose optimization techniques that improve the throughput of parallel GC on NUMA architecture. The optimizations were done to the GC algorithm to improve the locality of objects and GC threads, but not to the load balance in the parallel GC.

Studies find that non-scalability can exist in single socket architecture also. Du Bois et al. [7] evaluate the time distribution between application threads on different numbers of CPU cores with the bottle graph, a visualization tool for multi-threaded program performance. The GC takes longer when more than 2 or 4 GC threads are employed, where the number of GC threads and application threads matches the number of CPU cores. Their explanation is that with more application threads, the heap region fills up faster , resulting in more frequent GC invocations and longer GC execution time.

Qian et al. [23] evaluate the scalability of both mutation and parallel GC on NUMA architecture, because the Java applications' execution time is spent on both mutation and GC. In their work, the change of the object lifespan is found as one of the reasons that the parallel GC performs worse on more CPU cores. The GC efficiency is reduced because the object lifespan becomes longer, and the longer object lifespan is caused by more parallelized mutation threads allocating and using objects on the heap concurrently.

The Garbage First algorithm is a concurrent garbage collection algorithm, partitions the heap into fixed-size small regions [6], which is designed for applications with large heap requirement and many parallel mutation threads. The collection work can be done to any regions, so the pause time is short. Parallel GC threads with work stealing for load balancing is used for collection, so the results of our work are applicable to this collector. Wu et al. [27] propose and evaluate a synchronization-free parallel marking algorithm, called task-pushing. Marlow et al. [19] describe a technique that enables each CPU core to perform minor garbage collection independently on multicore architectures. Their work is done with the Haskell compiler, and the general idea is similar to the work by Gidra et al. Xian et al. [29, 30, 28] propose and implement a new generational collection algorithm that invokes collection at different times instead of waiting until the heap region is full to improve the performance of garbage collection.

Other research efforts explore the new methodologies for Java heap management on NUMA architecture [18, 22, 26]. They are good complements to the existing NUMA-aware young generation allocation/GC in OpenJDK.



(a) avrora on NUMA

(b) pmd on NUMA

(c) avrora on UMA

(d) pmd on UMA

Figure 10: The scalability comparison of three work stealing strategies

8. CONCLUSION

Work stealing has been employed for balancing the workload in parallel garbage collection. However, it does not deliver the expected improvement and worse, can degrade the GC performance as compared to implementations employing no stealing. We identified several factors that limit the efficiency of work stealing. These include futile stealing attempts, excessive calls to functions that locate stealing targets, and too many stealing tasks especially toward the end of the parallel phases when GC work is nearly done. Based on this analysis, we implemented a better work stealing mechanism called SmartStealing. According to the evaluation results on both on the UMA and NUMA architectures, SmartStealing can improve the GC performance for most of the applications to 50.4% in average and 78.7% at most.

The scalability of the GC performance is also impacted by the work stealing according to the experiment results that the GC performance is stable without work stealing when more GC threads are used. The work stealing makes the parallel phase in the GC take longer time with more GC threads, and this is would be our future work to improve the scalability of parallel GC.

Acknowledgment

This material is based on research sponsored by DARPA and ARO under agreement numbers FA8750-14-2-0053 and W911NF-13-1-0154, 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] Garbage collection roots. http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.mat.ui.helpFconceptsFgcroots.html.

[2] Memory management in the java hotspot virtual machine. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf.

[3] 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.

[4] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 557–558. ACM, 2010.

[5] K.-Y. Chen, J. M. Chang, and T.-W. Hou. Multithreading in java: Performance and scalability on mul-

ticore systems. *Computers, IEEE Transactions on*, 60(11):1521–1534, 2011.

[6] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48. ACM, 2004.

[7] 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.

[8] 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.

[9] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[10] 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.

[11] 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.

[12] 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.

[13] E. Helin. Improving load balancing during the marking phase of garbage collection. 2012.

[14] D. Hendler and N. Shavit. Work dealing. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 164–172. ACM, 2002.

[15] B. Iyengar, E. Gehringer, M. Wolf, and K. Manivannan. Scalable concurrent and parallel mark. *ACM SIGPLAN Notices*, 47(11):61–72, 2013.

[16] R. Iyer, M. Bhat, L. Zhao, R. Illikkal, S. Makineni, M. Jones, K. Shiv, and D. Newell. Exploring small-scale and large-scale cmp architectures for commercial java servers. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 191–200. IEEE, 2006.

[17] J. Levon, P. Elie, et al. Oprofile. *A system-wide profiler for Linux systems. Homepage: http://oprofile. sourceforge. net*, 2006.

[18] Z. Majo and T. R. Gross. Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In *ACM SIGPLAN Notices*, volume 46, pages 11–20. ACM, 2011.

[19] S. Marlow and S. Peyton Jones. Multicore garbage collection with local heaps. In *ACM SIGPLAN Notices*, volume 46, pages 21–32. ACM, 2011.

[20] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In *ACM Sigplan Notices*, volume 44, pages 45–54. ACM, 2009.

[21] C. E. Oancea, A. Mycroft, and S. M. Watt. A new approach to parallelising tracing algorithms. In *Proceedings of the 2009 international symposium on Memory management*, pages 10–19. ACM, 2009.

[22] T. Ogasawara. Numa-aware memory manager with dominant-thread-based copying gc. In *ACM SIGPLAN Notices*, volume 44, pages 377–390. ACM, 2009.

[23] 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.

[24] K. Shiv, R. Iyer, M. Bhat, R. Illikkal, M. Jones, S. Makineni, J. Domer, and D. Newell. Addressing cache/memory overheads in enterprise java cmp servers. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 66–75. IEEE, 2007.

[25] J. SPEC. Client2008 suite. *URL: https://www.spec.org/ jvm2008/*.

[26] M. M. Tikir and J. K. Hollingsworth. Numa-aware java heaps for server applications. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 108b–108b. IEEE, 2005.

[27] M. Wu and X.-F. Li. Task-pushing: a scalable parallel gc marking algorithm without synchronization operations. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[28] F. Xian, W. Srisa-an, C. Jia, and H. Jiang. As-gc: An efficient generational garbage collector for java application servers. In *ECOOP 2007–Object-Oriented Programming*, pages 126–150. Springer, 2007.

[29] 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.

[30] 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.