Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems

Kun Suo[1], Jia Rao[1], Hong Jiang[1] and Witawas Srisa-an[2]
[1] *University of Texas at Arlington, {kun.suo, jia.rao, hong.jiang}@uta.edu*
[2] *The University of Nebraska–Lincoln, witty@cse.unl.edu*

ABSTRACT

The proliferation of applications, frameworks, and services built on Java have led to an ecosystem critically dependent on the underlying runtime system, the Java virtual machine (JVM). However, many applications running on the JVM, e.g., big data analytics, suffer from long garbage collection (GC) time. The long pause time due to GC not only degrades application throughput and causes long latency, but also hurts overall system efficiency and scalability.

In this paper, we present an in-depth performance analysis of GC in the widely-adopted HotSpot JVM. Our analysis uncovers a previously unknown performance issue – the design of dynamic GC task assignment, the unfairness of `mutex` lock acquisition in HotSpot, and the imperfect operating system (OS) load balancing together cause loss of concurrency in Parallel Scavenge, a state-of-the-art and the default garbage collector in HotSpot. To this end, we propose a number of solutions to these issues, including enforcing GC thread affinity to aid multicore load balancing and designing a more efficient work stealing algorithm. Performance evaluation demonstrates that these proposed approaches lead to the improvement of the overall completion time, GC time and application tail latency by as much as 49.6%, 87.1%, 43%, respectively.

CCS CONCEPTS

• **General and reference → Performance**; • **Software and its engineering → Garbage collection**; • **Computer systems organization → Multicore architectures**;

KEYWORDS

Java virtual machine, Garbage collection, Performance, Multicore.

ACM Reference Format:
Kun Suo[1], Jia Rao[1], Hong Jiang[1] and Witawas Srisa-an[2] [1] *University of Texas at Arlington, {kun.suo, jia.rao, hong.jiang}@uta.edu* [2] *The University of Nebraska–Lincoln, witty@cse.unl.edu* . 2018. Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, Porto, Portugal, 15 pages. https://doi.org/10.1145/3190508.3190512

EuroSys '18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190512

1 INTRODUCTION

Due to its ease of use, cross-platform portability, and wide-spread community support, Java is becoming popular for building large-scale systems. Many distributed systems, such as Cassandra [2], Hadoop [11], Kafka [17], and Spark [40], are written in Java. Furthermore, there is also a steady trend towards adopting similar managed programming languages in high performance computing (HPC) [28, 42, 48]. Garbage collection (GC) is a crucial component of the automatic memory management in managed runtime systems, e.g., the Java Virtual Machine (JVM). It frees up unreferenced memory in the heap such that programmers do not need to concern about explicit memory deallocation. However, many studies [8, 9, 29] have shown that the widely adopted, throughput-oriented GC design suffers from suboptimal performance and poor scalability on the multicore systems with large memory and high core count.

Throughput-oriented GC pauses mutators, i.e., application threads, during GC to avoid expensive synchronizations between the GC and mutator threads. This period is called a stop-the-world (STW) pause. Since mutators cannot make progress during a STW pause, GC time can contribute to a non-trivial portion of application execution time. Previous work has shown that GC can take up to one-third of the total execution time of an application [7, 8]. It can even account for half of the processing time in memory-intensive big data systems [10, 29]. The exceedingly long GC time hurts system throughput and incurs unpredictable and severely degraded tail latency in interactive services [6, 23].

Parallel GC employs multiple GC threads to scan the heap and is designed to exploit hardware-level parallelism to reduce STW pause time. However, many studies have reported inefficiency and poor scalability of parallel GC on multicore systems. Existing studies [8, 9, 12, 34, 44] focus on optimizing the parallel GC algorithm in the JVM and assume that the underlying operating system (OS) provides the needed parallelism to execute parallel GC. There has been much research on analyzing the scalability of multi-threaded applications based on this assumption. We found that OS thread scheduling, particularly multicore load balancing, can have substantial impact on parallel GC performance. Our experiments with OpenJDK 1.8.0 and the Parallel Scavenge (PS) garbage collector revealed that many representative Java applications, including programs from the *DaCapo*, *SPECjvm2008*, and *HiBench* big data benchmarks, are unable to fully exploit multicore parallelism during GC. The main culprit is the uncoordinated design of the JVM and the underlying multiprocessor OS. On the one hand, modern OSes have complex load balancing algorithms due to the consideration of scalability, data locality, and energy consumption. Depending on different types of workloads, the OS thread scheduler needs to

strike a balance between grouping threads on a few cores and distributing them on many cores. On the other hand, the JVM, which assumes perfect OS load balancing, has its own design for efficient load balancing among GC threads and synchronization primitives used within the JVM, e.g., `mutex`.

In this paper, we identify two vulnerabilities in the HotSpot JVM and Parallel Scavenge due to the lack of coordination with OS-level load balancing. **First**, Parallel Scavenge implements dynamic GC task assignment to balance load among GC threads, but uses an unfair `mutex` lock to protect the global GC task queue. Although the unfairness is necessary for minimizing locking latency and believed to be harmless to GC performance, it inadvertently limits the concurrency in parallel GC when the underlying OS load balancing is "imperfect" and some GC threads are stacked on the same core. In this case, one or a few GC threads, which are able to continuously fetch GC tasks, will block other GC threads, leaving much of multicore parallelism unexploited. Since the unfair `mutex` implementation is also used for synchronizing VM threads and mutators, this problem may also exist in many user space Java applications. **Second**, to further balance GC load, an idle GC thread steals work from a randomly selected GC thread. A steal attempt fails if the selected GC thread has no extra work to be stolen. This lack of coordination between the JVM and the multicore OS causes the heuristics that guide work stealing to be ineffective, which delays the termination of the GC.

To address these vulnerabilities, we propose two optimizations in the HotSpot VM. Through an in-depth analysis of the effect of unfair locking on GC performance and the evaluation of two fixes to the unfairness issue in the JVM `mutex`, we find that GC thread affinity, which dynamically binds GC threads to separate cores based on CPU load, is effective in preventing load imbalance among GC threads. To address the inefficiency in GC work stealing, we devise an adaptive stealing policy that dynamically adjusts the number of steal attempts according to the number of active GC threads and improves steal success rate using a semi-random stealing algorithm. Our experiments with industry-standard benchmarks, *DaCapo* and *SPECjvm2008*, and two real-world Java applications, *Cassandra* database and *HiBench* big data benchmarks, show up to 49.6%, 87.1% and 43% improvement on application execution time, GC time and request tail latency, respectively, due to our optimizations on parallel GC. To summarize, this paper makes the following contributions:

- **In-depth analysis of GC performance in multicore systems.** We leverage comprehensive GC profiling, knowledge of OS scheduling and thread synchronization to identify vulnerabilities in the Parallel Scavenge and the HotSpot JVM that can inflict a loss of concurrency during parallel GC. The resulted load imbalance significantly prolongs the STW pause time.
- **Proposing two optimizations to address GC load imbalance.** We discuss our attempts to addressing GC load imbalance and propose a dynamic GC load balancing scheme with coordination between the JVM and the OS kernel. We further improve GC load balancing by designing an adaptive and semi-random work stealing algorithm inside the JVM.
- **Comprehensive evaluations of the proposed optimizations.** Our evaluations on *DaCapo*, *SPECjvm2008*, *Cassandra*, and *HiBench* show considerable and consistent improvement on parallel GC. We also demonstrate that our optimizations improve the performance on application scalability, various heap configurations as well as in complex application execution environments,

The rest of this paper is organized as follows. § 2 introduces the design of Parallel Scavenge, monitor-based synchronization in the HotSpot JVM, and explains how load balancing works in Linux CFS. § 3 analyzes the root causes of GC load imbalance and inefficient GC work stealing and § 4 proposes two optimizations to addressing these issues. § 5 presents experimental results and analysis. § 6 reviews the related work and § 7 concludes this paper.

2 BACKGROUND

2.1 Parallel Scavenge

Garbage Collection is the process of automatically freeing objects that are no longer referenced by application threads (mutators). It scans root references in the heap and records references that are reachable during the scan. Objects with unreachable references are regarded as garbage and are reclaimed in a sweep phase. Parallel Scavenge uses a stop-the-world design, which pauses mutator threads until GC completes. The collection involves three phases: *initialization phase*, *parallel phase*, and *final synchronization phase* [8]. In the initialization phase, the *VM thread* ensures that all mutator threads are suspended before waking up GC threads. After the GC threads become live, the VM threads sleep and wait for the final phase. Collection is performed in the parallel phase, in which the `GCTaskManager` creates and adds GC tasks into the `GCTaskQueue` from where multiple GC threads can fetch and execute them in parallel. With the help of the global task queue, Parallel Scavenge implements dynamic task assignment among GC threads (Section 2.2).

Parallel Scavenge performs generational garbage collection [1] by dividing the heap into multiple generations: *young*, *old*, and *permanent* generation. The young generation is further divided into one eden space and two survivor spaces, i.e., from-space and to-space. When the eden space is filled up, a minor GC is performed. Referenced objects in eden and from survivor space are moved to the to survivor space, and unreferenced objects are discarded. After a minor GC, the eden and the from space are cleared, and objects survived in the to space have their age incremented. After surviving a predefined number of minor GCs, objects are promoted to the old generation. Similarly, as the old generation is filled up, a major GC is triggered to free space in the old generation. Both minor and major GCs obtain tasks from `GCTaskQueue` except that `GCTaskManager` prepares different GC tasks for them. Among GC tasks, steal tasks are used to balance load between GC threads and are always placed after normal GC tasks in `GCTaskQueue`. GC threads that have fetched steal tasks attempt to steal work from other GC threads. When all GC threads complete the parallel phase and suspend themselves, the VM thread is woken up, entering the final synchronization phase. After resizing the generations based on the feedback of recently completed GCs, the VM thread wakes up the mutators and suspends itself until the next GC.

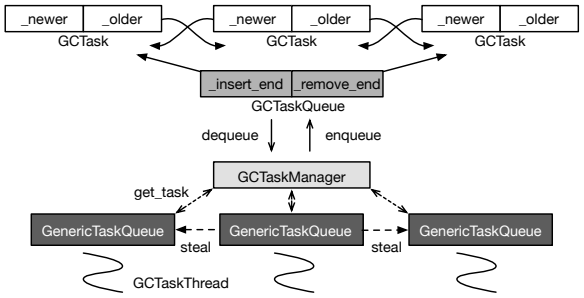

Figure 1: Dynamic task assignment in Parallel Scavenge.

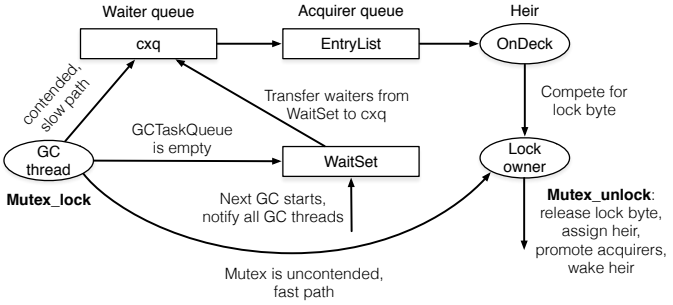

Figure 2: Native monitor in the HotSpot JVM.

2.2 Dynamic GC Task Assignment

Figure 1 shows the implementation of dynamic GC task assignment during the parallel phase of Parallel Scavenge. At the beginning of the parallel phase, `GCTaskManager` adds various types of GC tasks, e.g., `OldToYoungRootTask`, `ScavengeRootsTask`, `ThreadRootsTask`, and `StealTask`, to the `GCTaskQueue`. As these tasks may contain different amounts of work, the load assigned to GC threads can be unbalanced. Dynamic task assignment, which only sends a task to a GC thread when it requests one, helps resolve the imbalance as GC threads assigned with smaller tasks would fetch more. To prevent concurrent access to `GCTaskQueue`, `GCTaskManager` is implemented as a monitor, which can only be owned by one GC thread at a time. GC threads keep attempting to fetch (i.e., `get_task`) and execute a task each time. Multiple GC threads are synchronized by a monitor-based GC task `manager`. If the queue is empty, i.e., all GC tasks have been completed, a GC thread suspends itself to the `WaitSet` in the monitor. Threads sleeping in the `WaitSet` can later be woken up when new tasks are added to the task queue, i.e., when the next GC begins. The waking GC threads compete for the `mutex` lock before they can dequeue a GC task.

2.3 Work Stealing among GC Threads

To further balance load, a GC thread can steal work from another thread if it would otherwise stay idle. Once a GC task is fetched from the task queue, a GC thread divides it into many fine-grained tasks and pushes them into a local task queue, i.e., `GenericTaskQueue` in Figure 1. Such finer-grained tasks can be stolen by others. For example, a root task in the young-generation collection pushes every reference it accesses in the object graph to the local breadth-first-traversal queue, in which each reference leading to a sub-graph is a fine-grained task.

Parallel Scavenge places steal tasks, one for each GC thread, after ordinary GC tasks in `GCTaskQueue`. Therefore, if no ordinary tasks are available in the queue, GC threads fetch steal tasks and start work stealing. A GC thread enters the final synchronization phase when `GCTaskQueue` is empty and there is no task to be stolen from other GC threads. Parallel Scavenge uses a distributed termination protocol to synchronize GC threads. After $2 * N$ consecutive unsuccessful steal attempts, a GC thread enters the termination procedure, where N is the number of GC threads. [1] It atomically increments a global counter `_offered_termination` to indicate termination. If the counter reaches N, all GC threads have terminated and the parallel phase ends. While in the termination protocol, a GC thread periodically peeks if there are any root tasks available from any of the GC threads. If so, it decrements the counter and returns back to stealing. The core of a steal attempt is the function `steal_best_of_2` that selects two randomly chosen GC threads and steals tasks from the one with the longer queue [27].

[1] HotSpot uses a heuristic to determine the number of GC threads: $N = (ncpus \leq 8)?ncpus : 3 + ((ncpus * 5)/8)$, here $ncpus$ denotes the number of CPU cores.

2.4 The Implementation of Monitor in HotSpot

Monitor is a synchronization mechanism that contains a condition variable and its associated `mutex` lock. It allows threads to have mutual exclusive access to a shared data structure and to wait on a certain condition. Figure 2 shows the structure of the native monitor in HotSpot. Parallel Scavenge implements `GCTaskManager` as a monitor to protect `GCTaskQueue`. When `GCTaskQueue` is empty, either before the first GC or at the end of the previous GC, all GC threads sleep in `WaitSet`. `GCTaskManager` notifies and wakes up all GC threads when the next GC begins.

The critical design in monitor is the `mutex` lock, which should strike a balance between efficiency and scalability. To this end, HotSpot implements two paths, fast and slow, for lock acquisition. A thread acquires the ownership of a `mutex` by changing the `LockByte` in the `mutex` from zero to non-zero using an atomic compare-and-swap (CAS) instruction. On the fast path, a thread first attempts to CAS the `LockByte`. If the `mutex` is not contended, lock acquisition is successful. Otherwise, the thread turns to the slow path. Although a CAS fast path offers low locking latency for a small number of threads, it incurs considerable cache coherence traffic on a many-core system with a large number of threads [20].

The slow path is designed for scalability. It contains two separate queues for lock contenders: `cxq` and `EntryList` and two internal lock states: `OnDeck` and `owner`. Recently-arrived threads push themselves onto `cxq` if the fast path fails. In addition, `GCTaskManager`, at the beginning of each GC, transfers sleeping GC threads from the `WaitSet` of the monitor to `cxq`, letting waking GC threads compete for the `mutex`. `Owner` is the current lock holder and `OnDeck` is the thread selected by the `owner` as the presumptive heir to acquire the lock. The `OnDeck` thread is promoted from the `EntryList`. If `EntryList` is empty, `owner` moves all threads on `cxq` to `EntryList`. Both queue promotion and heir selection are performed by the lock `owner` when it unlocks the `mutex`.

This slow path is efficient for highly contended `mutex`. It throttles concurrent attempts to acquire the lock from a large number of

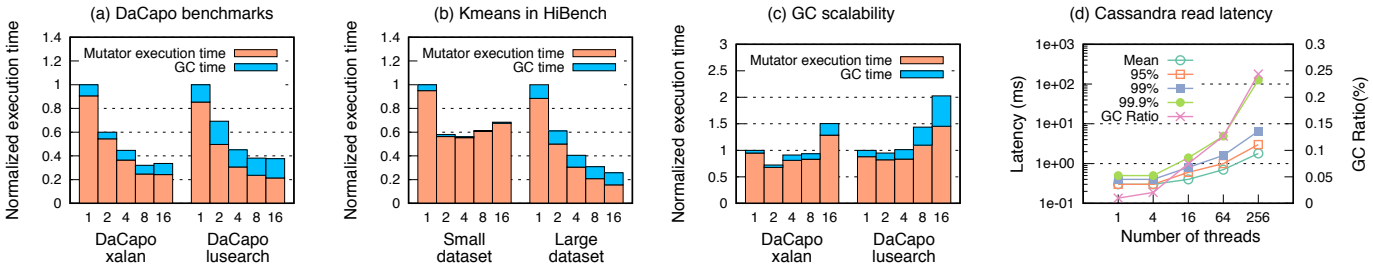

Figure 3: The impact of GC on application performance.

threads. For example, GC threads are transferred from `WaitSet` to `cxq` without being woken up to avoid severe contention from multiple CAS attempts. Furthermore, threads on `cxq` or `EntryList` are in the sleep state and not allowed to attempt lock acquisition. HotSpot ensures that there can be at most one `OnDeck` thread. Thus, at any time, there are at most three (types of) contenders on the lock: the `OnDeck` thread, the `owner` that just released the lock, newly arrived thread(s) which has not been placed on `cxq`. To avoid the lock-waiter preemption problem [32], in which a thread supposed to acquire the lock is preempted, delaying other waiters, HotSpot uses a *competitive handoff* policy. Instead of directly passing the lock from the `owner` to the `OnDeck` thread, the `owner` wakes up the `OnDeck` thread and lets it compete for the lock by itself. As such, even if `OnDeck` is preempted, other threads are still able to acquire the lock through the fast path. Although the above `mutex` design provides excellent throughput, it sacrifices short-term fairness: 1) it allows the `owner` thread to re-acquire the `mutex` lock, possibly causing starvation to the lock waiters on `cxq` and the `OnDeck` thread; 2) newly-arrived threads can bypass the queued lock waiters. We will show in § 3 that the short-term unfairness can cause severe inefficiency in the parallel phase of Parallel Scavenge.

2.5 Linux Load Balancing

Load balancing is a critical component in an OS scheduler. By evenly distributing threads on all cores, it minimizes the average queuing delay on individual cores and exploits the parallelism on multicore hardware. However, load balancing has become very complex in modern OSes due to the consideration of overhead, scalability, data locality, and power consumption. In this section, we describe the load balancing algorithm in Linux's Completely Fair Scheduler (CFS), the widely used OS scheduler in production systems.

Due to scalability concerns, CFS uses per-core run queues on a multicore machine. Individual cores are responsible for time sharing the CPU among multiple threads. Load balancing is implemented by means of thread migration across cores. Overall, CFS tracks load on each core and transfers threads from the most loaded core to the least loaded. Since thread migrations between cores require inter-core synchronization, load balancing should not be performed too frequently to avoid high overhead. In general, there are three scenarios that trigger load balancing: 1) a core becoming idle for the first time will attempt to steal runnable threads from the run queue of a busy core; 2) a core periodically runs the balancing algorithm; 3) a waking thread can be migrated from its current core to the idlest core in the system.

Load balancing can be ineffective for several reasons. First, Linux only migrates "runnable" threads between cores. Thus, GC threads that frequently sleep may miss either idle balancing (scenario 1) or periodic load balancing (scenario 2). In addition, the load balance interval in CFS is coarse grained compared to the length of GC tasks. For example, the default interval for periodic load balancing between two hyperthreads is 64*ms* and the interval increases (multiply by 2) as the distance between the CPUs increases. In comparison, for most applications, the GC should complete within a few hundreds of milliseconds to avoid a long pause of the application threads. Therefore, each individual GC task typically lasts a few tens or hundreds of microseconds. Third, CFS avoids waking up idle cores that are in a deep sleep state for load balancing to save energy. In any of these circumstances, multiple GC threads can be stacked on a few cores even when there are idle cores in the system. Therefore, load balancing in CFS is most effective with workloads with a stable degree of parallelism but fails to function properly with GC threads that exhibit dynamic parallelism.

3 ANALYSIS OF INEFFICIENT PARALLEL GC

This section presents an in-depth analysis of the performance of Parallel Scavenge on a multicore machine. We first show the poor scalability of parallel GC and its impact on application performance. Then, we attribute the suboptimal GC performance to load imbalance among GC threads and inefficient stealing.

3.1 Parallel GC Performance and Scalability

We selected four workloads, two scalable workloads *xalan* and *lusearch* from the *DaCapo* benchmarks [5], *kmeans*, a well-known clustering algorithm for data mining from the *HiBench* big data benchmarks [14], and *Cassandra*, a distributed NoSQL database. For the traditional JVM workloads, such as *xalan* and *lusearch*, the JVM heap size was set to three times of the minimum heap requirement [13]. *Kmeans* and *Cassandra* had a heap size of 4GB and 8GB, respectively. All benchmarks were executed on Linux 4.9.5, OpenJDK 1.8.0 and Parallel Scavenge. The experiments were tested on a Dell PowerEdge T430 with dual Intel 10-core processors. Details of the testbed and benchmark settings can be found in Section 5.1.

Figure 3 (a) shows the performance of *xalan* and *lusearch* with various numbers of mutator threads and a breakdown of mutator execution time and GC time. Parallel Scavenge sets the number of GC threads to 15 on our 20-core machine. As shown in Figure 3 (a),

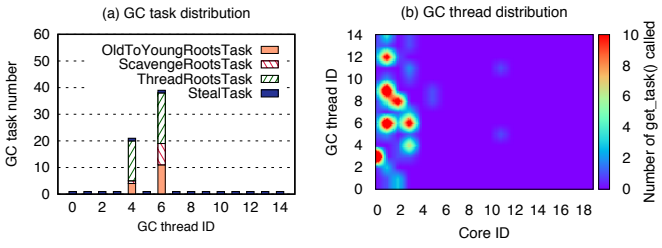

Figure 4: Task and thread load imbalance in *lusearch* GC.

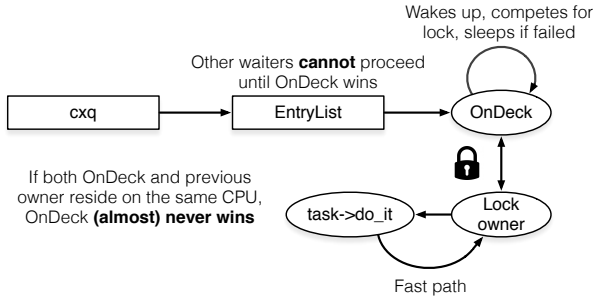

Figure 5: The vulnerability of dynamic task assignment and unfair `mutex` lock in multicore systems.

in *xalan* and *lusearch*, mutator time dropped as the number of mutators increased and thus GC time became more significant in the overall execution time. For instance, GC contributed to 43.2% of the total time in the case of 16 mutator threads, incurring unacceptable overhead to application performance. Next, we evaluated the performance of *kmeans* in Spark. Figure 3 (b) shows the performance with a varying number of mutators and two input sizes: small and large. Similar to the *DaCapo* results, the ratio of GC time increased as mutator time decreased. In addition, the large dataset incurred much higher GC overhead compared to the small dataset.

In Figure 3 (c), we fixed mutator threads to 16 and varied the number of GC threads to study GC scalability. For both *xalan* and *lusearch*, parallel GC scaled poorly with increasing parallelism. The GC time even ramped up as the number of GC threads increased. Another observation is that mutators had prolonged execution time with more GC threads. It suggests that inefficient GC not only hurts JVM memory management but also influences mutator performance. Figure 3 (d) studies request latency in the *Cassandra* database. We used another client machine executing a varying number of threads to read one million records from the database. The figure shows that the request latency increased exponentially with more intensive client traffic. The ratio of GC time in the total execution time also climbed to 25%. As an STW collector, the increased parallel GC time can significantly prolong tail latency. In what follows, we identify the causes of the inefficient GC and its poor scalability.

3.2 Load Imbalance

We instrumented the HotSpot JVM to report two types of load information during parallel GC: 1) GC task distribution among GC threads and 2) GC thread distribution on CPU cores. We analyzed *lusearch* as it shows significant GC overhead and poor scalability in Figure 3. The number of mutator threads was set to 16 and the number of GC threads was automatically set by Parallel Scavenge to 15.

Task Imbalance. To monitor task distribution, we modified the constructor of GC tasks to log the GC thread ID on which a GC task is executed. According to the GC logs, parallel GC in *lusearch* is dominated by minor GC, which includes fifteen *OldToYoungRootsTasks*, nine *ScavengeRootsTasks*, thirty-four *ThreadRootsTasks* and fifteen *StealTasks*. Note that the number of *StealTasks* matches the number of GC threads. As shown in Figure 4 (a), GC tasks, except *StealTasks*, were unevenly distributed among GC threads. `GCTaskThread` 4 and 6 processed all root tasks during minor GC while others only ran *StealTasks*. We observed a similar task imbalance in other JVM benchmarks. *Lusearch* incurred around 198 minor GCs during execution. The GC logs showed severe task imbalance in most minor GCs, though the ID(s) of overloaded GC thread(s) varied in each GC. The result clearly shows that Parallel Scavenge failed to exploit the available parallelism (i.e., 15 GC threads) in GC. Note that GC threads are homogeneous in Parallel Scavenge and do not have bias in task assignment, which led us to the investigation of GC thread execution in the underlying OS.

Thread Imbalance. To monitor GC thread execution, we traced function `GCTaskManager::get_task` and recorded the number of times each GC thread successfully dequeued a GC task from `GCTaskQueue` and on which CPU the GC thread was running. Figure 4 (b) shows the execution of all GC threads of one minor GC in *lusearch*. It suggests that most GC threads were stacked on a few CPU cores while the remaining cores were idle. It is evident that multicore parallelism was not fully exploited. Figure 4 (b) also shows unbalanced GC task distribution with a few threads having fetched more tasks than others did.

Root cause analysis. As discussed in Section 2.5, Linux load balancing can be imperfect and temporarily place multiple GC threads on the same core. However, two critical questions remain unanswered: 1) *why is OS load balancing not effective during the entire GC?* 2) *why is time slicing/sharing on a single core not effective, otherwise stacking GC threads should have equal opportunities to fetch tasks and task imbalance should never occur?* We identified the reasons by monitoring the competitions between GC threads on the `mutex` lock that protects the `GCTaskQueue`. The GC log showed that throughout the GC, at any point in time, there were at most two GC threads (the `OnDeck` thread and the previous `owner` thread) actively competing for the `mutex` lock and the previous `owner` thread (almost) always won.

Figure 5 illustrates how the loss of concurrency in parallel GC develops. **First**, at the beginning of each GC, sleeping GC threads on the monitor's `WaitSet` are transferred to `cxq` and become waiters of the `mutex` lock. This is to prevent concurrent attempts on lock acquisition when all GC threads wake up at the same time. The monitor selects two threads at the head of `cxq` to be the lock `owner` and `OnDeck`, and the other threads remain blocked, not eligible for lock acquisition nor OS load balancing. Then, lock competition becomes a two-player game. **Second** and most importantly, the

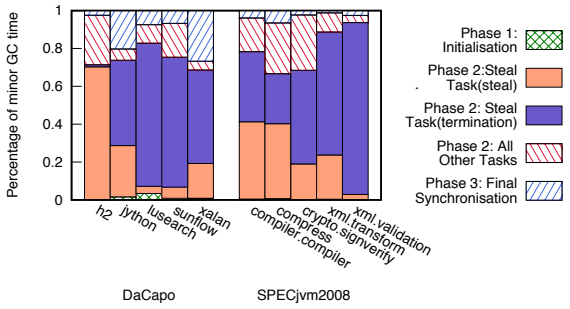

Figure 6: The decomposition of minor GC time.

Benchmark	Total	Failure	Failure rate
h2	237983	166008	69.8%
jython	75036	66318	88.4%
lusearch	117383	108308	92.3%
sunflow	45648	34695	76.0%
xalan	22783	19303	84.7%
compiler.compiler	726920	274043	37.7%
compress	29348	26513	90.3%
crypto.signverify	64493	60388	93.6%
xml.transform	457198	341555	74.7%
xml.validation	651475	188189	28.9%

Table 1: The total and failed steal attempts in steal_best_of_2().

competition is unfair if the two threads are stacked on the same CPU. As shown in Figure 5, after the `owner` releases the lock and wakes up `OnDeck`, it executes the GC task it fetched from `GCTaskQueue`. Once the task is completed, the previous `owner` thread executes `get_task` again, attempting to fetch another GC task and acquire the `mutex` lock. The fast lock acquisition path allows the previous `owner` to bypass waiters in `cxq` and `EntryList` and directly acquire the lock. If the two threads are on the same core, the `OnDeck` thread may never acquire the lock.

Most OS schedulers avoid frequent thread context switching on a CPU and guarantee that a thread can run for a minimum time before it is preempted. Therefore, after releasing the lock and waking up the `OnDeck` thread, the previous `owner` thread may continue to run on CPU if it has not used up its minimum time quantum. In this case, the waking `OnDeck` thread would fail to preempt the `owner` thread and be placed by the scheduler onto the CPU run queue as a runnable thread. Since the `OnDeck` misses the wakeup momentum to preempt the `owner` thread, it has to wait for a whole time slice before being scheduled. At the time the `OnDeck` thread is scheduled, if the `owner` thread has re-acquired the lock, the `OnDeck` thread would go to sleep again. This cycle repeats and the `OnDeck` thread may never acquire the lock until the `owner` thread depletes all GC tasks in the `GCTaskQueue`.

The stacking of GC threads will happen almost every time. When GC threads are first created by the JVM, they are spawned on one core. Since the GC task queue is empty at the launch time of the JVM, all GC threads will immediately block until the first GC begins. OS schedulers do not balance blocked threads, thus all GC threads are stacked on one core when the first GC starts, relying on OS load balancing to resolve the stacking. There are two practical obstacles to effectively balancing stacked GC threads.

First, during lock contention, there are at most two active GC threads, i.e., the `owner` thread and the `OnDeck` thread, that are eligible for load balancing. However, if the `OnDeck` cannot acquire the lock and remain in a blocked state, load balancing will not take effect as there is no runnable thread to move. It is possible to decrease the minimum thread runtime to increase the chance of the `OnDeck` thread to preempt the `owner` thread, i.e., setting a smaller value for `sched_min_granularity_ns` in CFS. However, the tuning of CFS (i.e., setting the minimum thread runtime to 100μs) does not mitigate the unfairness in lock acquisition. At the beginning of each GC, the `OnDeck` thread is unlikely able to preempt the `owner` thread regardless of the minimum runtime as the `owner` also just woke up. After the first failed attempt, the `OnDeck` thread becomes runnable and has to wait a full time slice (12ms in CFS) to be scheduled. Once the `owner` thread is descheduled, its minimum runtime is reset. If the `OnDeck` thread fails to acquire the lock again, the vicious cycle of block, wakeup and failed lock acquisition continues.

Second, even if the `OnDeck` thread acquires the lock, the execution serialization still persists. The previously `OnDeck` thread becomes the `owner` thread and a sleeping lock waiter thread, which resides on the same CPU, will be promoted to `OnDeck`. Since the two new lock contenders are stacked on one CPU, the unfairness in lock acquisition still exists. Ideally, load balancing will be effective when all GC threads are active and visible to the load balancer. This requires that the non-critical section of GC, i.e., each GC task, be long enough to keep all GC thread busy at the time of load balancing. However, the amount of work in each GC task varies greatly, depending on the sparsity of the sub-graph reachable from a GC task. Some tasks can be quite small in a large heap. Therefore, it is almost impossible to keep all GC threads active all the time and thread stacking is inevitable. As a result, unfair locking in the JVM causes serialization among GC threads.

3.3 Ineffective Work Stealing

As discussed in Section 3.2, there exists significant load imbalance among GC threads. Next, we study the effectiveness of work stealing in addressing the imbalance among GC threads. Figure 6 shows the breakdown of minor GC time of some representative applications in *DaCapo* and *SPECjvm2008* [41]. We instrumented Parallel Scavenge to log the execution time of each GC stage. We further recorded detailed GC task completion time at each GC thread and divided the parallel GC phase into root task, steal task and steal termination. Note that the time breakdown in Figure 6 is aggregated among all GC threads. It is possible that when some threads were in steal termination, others were still executing root tasks. Thus, the GC time breakdown does not reflect the timeline of GC.

As shown in Figure 6, steal tasks dominate the total GC time in all benchmarks, which was also observed by Gidra et al. [7]. While a GC thread executes a steal task, it is either processing a task stolen from another thread or attempting a steal. In contrast, the time spent in the steal termination, during which terminated GC threads waiting for other active threads to synchronize at the barrier of the

Algorithm 1 Dynamic GC thread balancing

```
1:  Variable: The load on the i_th core L_i; the load degree of the i_th core
    D_i; the average load of across all cores L_avg; the number of cores with
    low load N_low; GC thread t_i.
2:  /* Mark CPU load as: high, normal, and low. */
3:  function MARK_CPU_LOAD_DEGREE
4:      for each core i do
5:          if L_i ≥ 2 * L_avg then
6:              D_i = high;
7:          else if L_i ≤ 0.5 * L_avg then
8:              D_i = low;
9:              N_low++;
10:         else
11:             D_i = normal;
12:         end if
13:     end for
14: end function
15:
16: /* Dynamically rebalance GC threads to avoid contentions */
17: function GC_THREAD_REBALANCE(t_i)
18:     if t_i's current CPU load degree is high then
19:         k = rand()%N_low;
20:         bind t_i to the k_th core in the low load CPU set;
21:     end if
22: end function
```

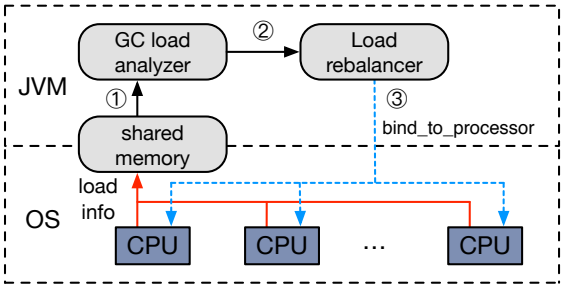

Figure 7: Proactive and dynamic GC load balance.

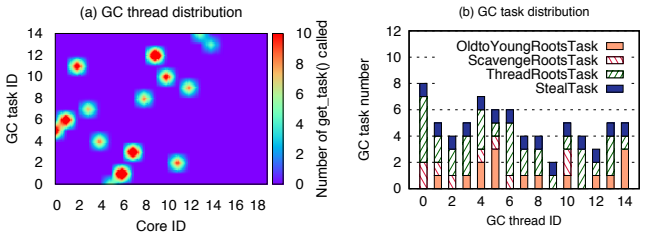

Figure 8: Improved thread and task balance in *lusearch* minor GC with 16 mutators.

final phase, is wasted and does not contribute any meaningful computation. Table 1 lists the total and failed steal in `steal_best_of_2`. Most benchmarks suffered high failure rate except *compiler.compiler* and *xml.validation*. Recall that `steal_best_of_2` selects the longer of two randomly chosen GC thread queues to steal. If the load is severely unbalanced, its performance can degrade to random stealing because most attempts return two empty queues. These observations motivated us to design a more efficient termination protocol and more effective stealing policy for steal tasks.

4 OPTIMIZATIONS

This section presents two optimizations of Parallel Scavenge to address its vulnerability and inefficiency in multicore systems (discussed in § 3). For each optimization, we describe its design and implementation, and evaluate its effectiveness on mitigating imbalance and wasteful stealing.

4.1 Addressing Load Imbalance

The culprits of load imbalance in parallel GC are ineffective OS load balancing of GC threads, unfair `mutex` acquisition, and dynamic task assignment that allows one or a few GC threads to deplete the GC task queue. To avoid re-designing the GC task model in Parallel Scavenge and changing dynamic task assignment, we explored optimizations to address unfair locking, such as disabling all fast paths in locking, enforcing fair (FIFO) `mutex` acquisition and allowing multiple active lock contenders. Unfortunately, without the help from the OS, these approaches either had no effect or led to degraded performance.

A simple approach to avoiding GC threads stacking is to disable the OS load balancing and pin GC threads to separate cores. `BindGCTaskThreadsToCPUs` has been included as a command line option in OpenJDK since version 1.4 but the backend function `bind_to_processor` was never implemented on Linux, Windows, or BSD. It was first proposed in specification JSR-282 but no agreement was made to provide the processor binding API due to lack of evidence for GC performance improvement and the difficulties to provide a generic API across various platforms [33]. The processor binding interface has been implemented in Solaris, but the benefits of binding was not well studied. To prove the necessity of GC thread affinity in parallel GC, we implemented this feature in OpenJDK 1.8.0 for Linux. When a `GCTaskThread` is created, it is bound to a core whose ID matches the thread ID.

Static binding avoids stacking GC threads on cores but may conflict with other workloads scheduled by the OS scheduler. To address this issue, we devise a GC thread balancing scheme (Algorithm 1) to rebalance GC threads to avoid contentions with other workloads. At the start of each parallel GC, a GC thread examines the load on its current CPU and binds to a different CPU if the current one experiences contention. We leveraged `load_avg` in the Linux kernel to measure the load on each CPU. Note that `load_avg` only measures the load of ready/running tasks but does not count sleeping threads. This explains why OS load balancing is not effective for stacking GC threads as most of them are in the sleep state. To avoid stacking GC threads during the rebalancing, we incorporated the load from sleeping threads into `load_avg` in the Linux kernel.

As Algorithm 1 shows, a CPU is considered to have a *high* load if its load is higher than two times of the average load across all CPUs while a *low* load is less than half of the system-wide average (line 4-9). Each GC thread checks the load of its current CPU at the start of each GC and rebinds to a randomly picked CPU with low load (line 14-15). Figure 7 shows the steps to rebalance GC

Algorithm 2 Adaptive and semi-random work stealing

```
1:  /* Adaptive termination protocol: only steal from active GC threads.*/
2:  Variable: The number of active GC threads N_live; the queue ID to
    steal q; the queue ID q_s in last successful steal attempt.
3:  function STEAL_TASK
4:      for Steal attempts less than 2 * N_live do
5:          q = STEAL_BEST_OF_2 (q_s)
6:          if q ≠ φ then
7:              q_s = q
8:              Steal from q and return
9:          end if
10:     end for
11:     q_s = φ
12:     Enter the termination protocol
13: end function
14:
15: /* A semi-random algorithm for queue selection */
16: Variable: The queue ID q_s in last successful steal attempt.
17: function STEAL_BEST_OF_2(q_s)
18:     Randomly select the first queue q_1
19:     if q_s ≠ φ and q_s is not empty then
20:         q_2 = q_s
21:     else
22:         Randomly select the second queue q_2
23:     end if
24:     if q_1 and q_2 are both empty then
25:         q_s = φ and return φ
26:     else
27:         return the longer of q_1 and q_2
28:     end if
29: end function
```

threads. The Hotspot JVM communicates with the Linux kernel via the `/proc` file system. The OS writes the run queue load of each CPU to the shared memory and the JVM reads the information when the `GCTaskManager` adds `GCTasks` to the `GCTaskQueue` and the GC threads are woken up (Step ①). The GC load analyzer determines whether it needs to rebind a GC thread to another CPU based on the load information (Step ②). If needed, our implemented `bind_to_processor` is used to rebind the GC thread (Step ③).

Besides the optimizations for GC thread balancing, we also made some modifications to improve the task-to-thread balance. The default `GCTask` has no task affinity even though the parameter `UseGCTaskAffinity` is enabled. We modified the constructor function of `OldToYoungRootsTask`, `ScavengeRootsTask` and `ThreadRootsTask` to include task affinity. When the `GCTaskThread` executes `get_task()`, `GCTaskManager` prefers to dequeue the task which has the affinity to that thread. If no task is found matching the affinity, `GCTaskManager` dequeues any task that is available.

We evaluated the effectiveness of the optimized thread and task balance design using `lusearch` from *DaCapo*, which was configured with 16 mutator threads and 15 GC threads. Figure 8 (a) shows that GC threads are evenly distributed on multiple cores. The warm temperature in the heatmap suggests that all GC threads were able to fetch tasks from `GCTaskQueue`. As shown in Figure 8 (b), GC thread and task affinity help mitigate load imbalance among GC threads. Compared to the case shown in Figure 4 (a), all GC threads

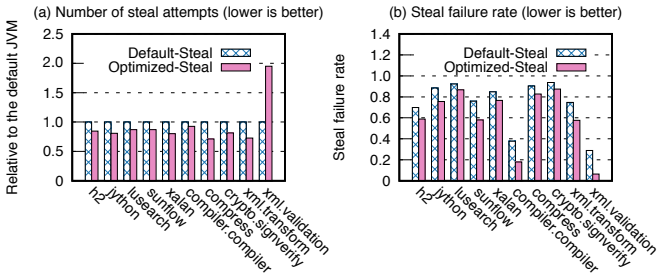

Figure 9: The optimized stealing algorithm reduces both steal attempts and failure rate. All workloads ran with 16 mutators.

are assigned with root tasks, showing much improved task balance among GC threads.

4.2 Addressing Inefficient Working Stealing

In Section 3.3, we identified two deficiencies of Parallel Scavenge's work stealing algorithm: 1) the distributed termination protocol is slow and incurs too many steal attempts; 2) the queue selection algorithm is not effective, leading to high steal failure rate. In the original design, a GC thread enters the termination protocol after experiencing $2 * N$ consecutive failed steal attempts, where N is the number of GC threads. For each attempt, it selects two random GC thread queues and steals from the longer one. We see two problems with such a design. First, the termination protocol requires $2 * N$ failures to end a GC threads regardless of how long the GC thread has been in the parallel GC. Towards the end of the parallel phase, most GC threads may have been in the termination protocol. Therefore, it is not necessary to wait for $2 * N$ failures to enter termination because there are only a few active threads, from which tasks can be stolen. Second, if load imbalance occurs, as we show in Figure 4 (a), work can be assigned to a few GC threads and the random stealing might be quite ineffective.

To address these two issues, we propose an adaptive and semi-random stealing algorithm, shown in Algorithm 2. We implemented a `FastParallelTaskTerminator` class in HotSpot to coordinate GC thread termination. It records the number of active GC threads (i.e., N_{live}) that are not yet in the termination protocol. As GC threads enter or exit the termination protocol, the active thread count is updated. A thread only steals from the pool of active GC threads (line 4-10). Accordingly, the criteria for thread termination becomes $2 * N_{live}$ consecutive failed steal attempts. The `steal_best_of_2` function was also modified to improve steal success rate. It memorizes the last queue from where the steal was a success and selects the same queue as one of the steal choices, given that the queue is not empty (line 19-20). Another queue is picked up randomly. Similarly, the longer of the two queues is chosen as the stealing target.

We evaluated the effectiveness of the optimized stealing algorithm using programs from *DaCapo* and *SPECjvm2008*. All programs ran with 16 mutator threads and 15 GC threads. As Figure 9 (a) shows, the optimized stealing reduced the total number of steal attempt for most of the benchmarks except *xml.validation*. Among these attempts, the portion of failed attempts, i.e., failure rate, also

dropped, as shown in Figure 9 (b). While the reduction on steal attempts or failure rate alone is not significant, the aggregate benefit is clear. For example, the number of steals for *xml.validation* increased by 95.1% while the failure rate dropped by 4.5x. As a result, the total number of failed attempts decreased by 56.8%, which indicates that the increased steal attempts contained mostly successful steals. Overall, the reduction on failed attempts ranged from 18.3% to 56.8%. As will be discussed in Section 5, the savings on futile steal attempts lead to improved GC performance.

5 EVALUATION

In this section, we present an evaluation of our optimized JVM using various micro and application benchmarks. We first study the effectiveness of our design on improving the overall application performance (§ 5.2) and on reducing GC time (§ 5.3), and compare our work with GC optimization in other papers. We then analyze the impact of improved GC on application scalability (§ 5.4), and investigate how much our design improves the performance of real-world applications (§ 5.5) and applications with different heap configurations (§ 5.6). Finally, we demonstrate the performance improvement in a multi-application environment (§ 5.7) and discuss the effect of simultaneous multithreading(§ 5.8).

5.1 Experimental Settings

Hardware. Our experiments were performed on a DELL PowerEdge T430 server, which was equipped with dual ten-core Intel Xeon E5-2640 2.6GHz processors, 64GB memory, Gigabit Network and a 2TB 7200RPM hard drive. Initially, simultaneous multithreading (SMT) was disabled to isolate the effect of our proposed optimizations from interference on sibling hyperthreads. The heuristic used to determine the number of GC threads is based on CPU count. Enabling SMT on our testbed results in a total of 40 logical cores and 28 GC threads. Since the testbed only has 20 physical cores, 16 GC threads would run on sibling hyperthreads, which could either have constructive or destructive impact on Java programs [15, 19, 26, 36]. We enable SMT and study its effect in Section 5.8. The machine was configured with the default power management and all cores ran at their maximum frequency. Turboboost, `mwait`, and low power C-States were also enabled. For database benchmarks, we used another machine in the same Ethernet as the client.

Software. We used Ubuntu 16.10 and Linux kernel version 4.9.5 as the host OS. All experiments were conducted on OpenJDK 1.8.0 with Parallel Scavenge as the garbage collector. If not otherwise stated, we set the number of mutators to 16 and the decision on the number of GC threads was left to Parallel Scavenge, which created 15 GC threads for our 20-core testbed. This setting ensured that the machine is under-provisioned and both the mutators and GC threads had access to sufficient multicore parallelism.

Benchmarks. We selected a subset of programs in the following benchmarks and measured their performance on the vanilla JVM and the one with our optimizations. The selected workloads are scalable workloads that benefit from parallel garbage collection. The heap sizes of these benchmarks were set to 3x of their respective minimum heap sizes [13, 46]. Details on heap configuration are listed in Table 2. To minimize non-determinism in multicore environments, each result was the average of 10 runs.

Benchmark	Suite	Heap size (MB)
h2	DaCapo	900
jython	DaCapo	90
lusearch	DaCapo	90
sunflow	DaCapo	210
xalan	DaCapo	150
compiler.compiler	SPECjvm	4000
compress	SPECjvm	2500
crypto.signverify	SPECjvm	2500
xml.transform	SPECjvm	4000
xml.validation	SPECjvm	4000
kmeans	HiBench	16384
wordcount	HiBench	16384
pagerank	HiBench	16384
Cassandra	Apache	8192

Table 2: Benchmark heap size.

- *DaCapo* is an open source client-side Java benchmark suite which consists of a set of real-world applications with non-trivial memory loads. The version of *DaCapo* we used in this paper is 9.12.
- *SPECjvm2008* is a benchmark suite for measuring the performance of a Java Runtime Environment, containing several real life applications focusing on core Java functionalities. The workloads mimic a variety of common general-purpose application computations.
- *HiBench* is a big data benchmark suite that contains a set of Hadoop, Spark and streaming workloads, including *kmeans*, *wordcount*, and *pagerank*, etc. The version we used in this paper is 6.0. We ran HiBench on Hadoop [11] 2.7.3 and Spark [40] 2.0.0 on a single node. The number of threads in the Spark executor was set to 16.
- *Cassandra* is an open-source distributed database management system designed to handle large amounts of data across many commodity servers. We use *cassandra-stress* to test the read and write latency. The version of *Cassandra* used in this paper was 3.0.10.

5.2 Improvement in Overall Performance

We first demonstrate the effectiveness of our optimizations on improving the overall performance of various Java applications. We adopted five benchmarks from *DaCapo* and *SPECjvm2008*, respectively, and compared their performance in the vanilla HotSpot JVM with that in our optimized JVM. Figure 10 (a) shows the execution times of five benchmarks from *DaCapo* under various settings. To quantify the performance improvement due to each optimization, we enabled one optimization at a time, GC affinity or optimized stealing, while disabling the other. The resulted JVMs are labeled as *w/ GC-affinity* and *w/ steal*, respectively. *Together* refers to the JVM with both optimizations enabled. Overall, GC thread affinity contributed more improvement compared to optimized stealing. For instance, the execution time of `sunflow` was improved by 30.4% and 17.5% due to thread affinity and optimized stealing, respectively. The performance improvement was more pronounced when both optimizations were taking effect. `sunflow` had a performance

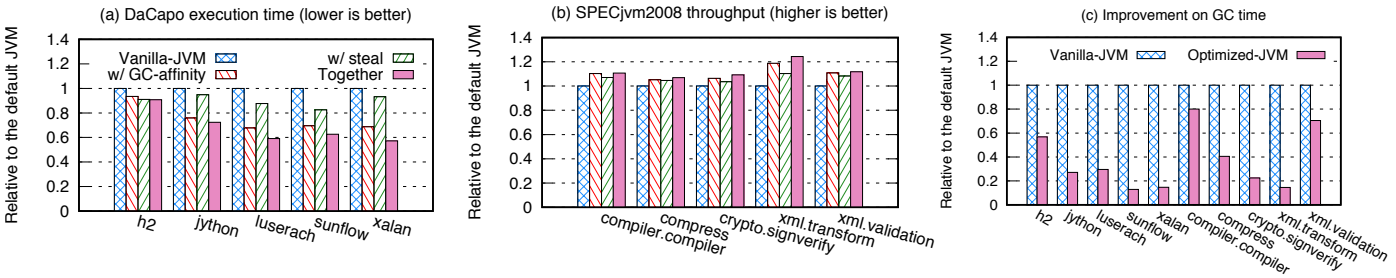

Figure 10: Overall and GC performance improvement on *DaCapo* and *SPECjvm2008*.

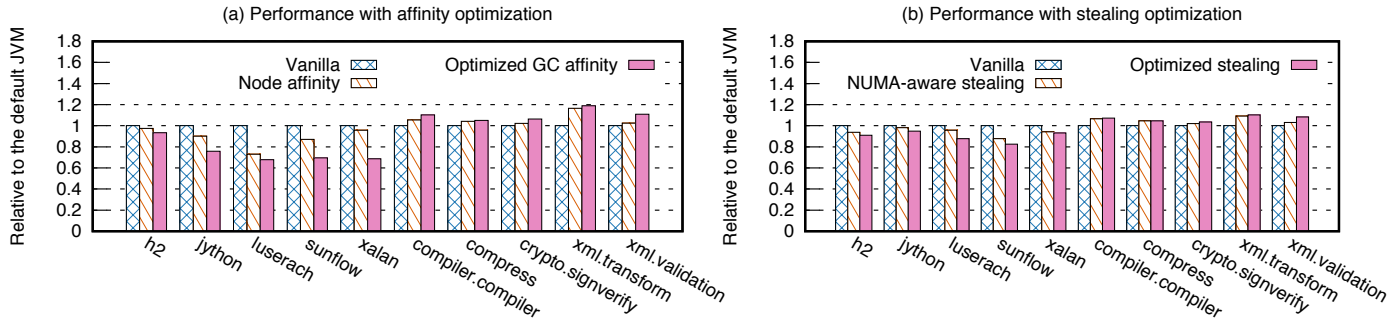

Figure 11: Performance comparison on *DaCapo* execution time (lower is better) and *SPECjvm2008* throughput (higher is better).

gain of 37.3% with *together*. Figure 10 (b) shows the throughput of *SPECjvm2008* benchmarks. Similarly, each optimization accelerated the operation speed of *SPECjvm2008* benchmarks. For instance, the throughput of *xml.transform* was improved by 18.9% and 10.3% with the thread affinity and optimized stealing, respectively. The combined optimizations improved its throughput by 24.3%. Note that the overall performance gain is lower than the aggregate gain of individual optimizations because each optimization improved the load balance, leaving less room to the other optimization for additional improvement.

Thread pinning has been used in HotSpot for improving data placement on non-uniform memory access (NUMA) machines. In particular, Gidra et. al. [8] proposed to segregate JVM heap in different NUMA nodes and restrict GC threads to only access objects in the local node. While the main purpose of segregated heap is to improve access balance across NUMA nodes, it imposes node affinity on GC threads, which could alleviate the GC thread stacking problem. In addition, segregated heap restricts work-stealing to GC threads within the same node, reducing the number of failed steal attempts before a GC thread enters the termination procedure. We ported the node affinity and NUMA-aware stealing proposed in [8] to OpenJDK 1.8.0 and compared their performance with our proposed optimizations.

Figure 11 (a) compares the performance of node affinity and our proposed dynamic thread affinity, using the vanilla JVM as the baseline. The results show that node affinity improved the overall performance compared to the vanilla JVM. The even distribution of GC threads to the two NUMA nodes in our testbed mitigated thread stacking, thereby improving concurrency during GC. However, our proposed thread affinity still outperformed the node affinity in most *DaCapo* benchmarks while achieving comparable performance in the *SPECjvm2008* benchmarks. It suggests that thread stacking can still happen within one NUMA node and the one-to-one thread-to-core binding is necessary for fully exploiting the potential of parallel GC.

Figure 11 (b) shows the results due to NUMA-aware stealing and our proposed adaptive stealing algorithm. From the figure, we can see that our approach achieved slightly better performance over NUMA-aware stealing. Note that we did not port NUMA-aware memory allocation proposed in [8]. The performance gain of NUMA-aware stealing was mainly due to reduced failed steal attempts. Since stealing is only allowed within a node, a GC thread stops stealing after $2 * N_{local}$ failed attempts, where N_{local} is the number of GC threads in a node. However, node affinity and NUMA-aware stealing share a common drawback – the static binding of GC threads to NUMA nodes and the restriction on where to steal GC tasks make the two approaches vulnerable to interference in a multi-user environment. In contrast, our approaches bind threads to the lightly loaded cores and steal from active threads, thereby more resilient to interference.

5.3 Improvement on Garbage Collection

Figure 10 (c) shows the performance improvement on garbage collection in *DaCapo* and *SPECjvm2008* benchmarks. Both optimizations were enabled and labeled as *optimized-JVM*. From this figure, we can see that our optimizations benefited all benchmarks and reduced GC time. The improvement in GC time ranged from 20% (*compiler.compiler*) to 87.1% (*sunflow*). In general, the benchmarks

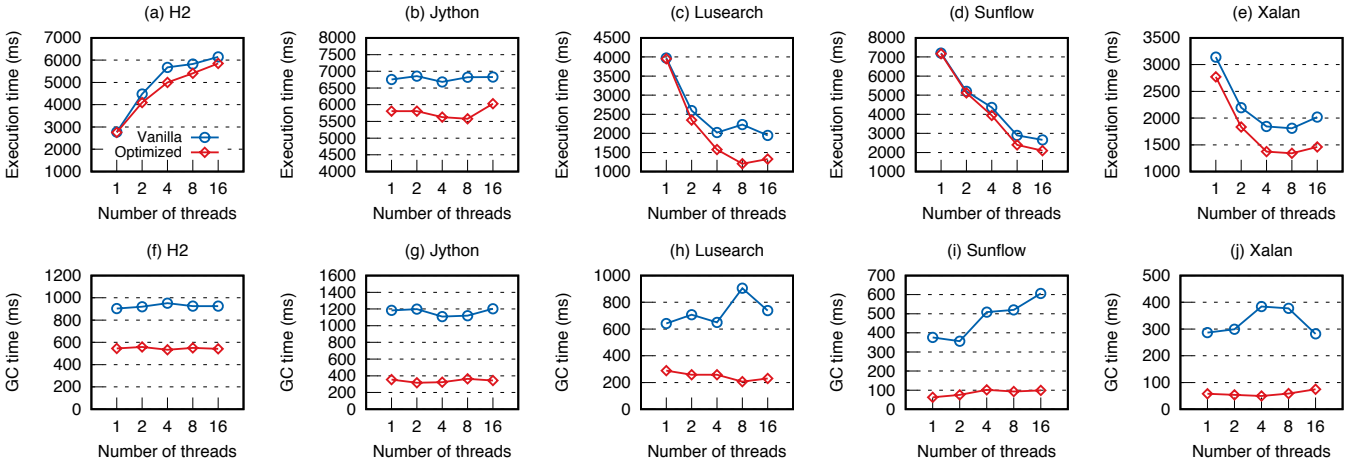

Figure 12: DaCapo: overall and GC scalability. (mutator varies from 1 to 16, GC thread number is 15)

that did not have as much improvement as others did were those had relatively low steal failure rate in Table 1, i.e., *compiler.compiler* and *xml.validation*. The low steal failure rate indicated good balance among GC threads, thereby less room for GC improvement. It confirms that load imbalance is the major inefficiency in GC on these benchmarks.

5.4 Scalability

GC load varies depending on the activities of mutator threads on the heap. Therefore, it is interesting to study how the number of mutators affects GC scalability and the performance impact of GC on overall application scalability. We let Parallel Scavenge configure the number of GC threads (i.e., 15 GC threads) and varied the number of mutators from 1 to 16. We evaluated two types of applications in *DaCapo*: non-scalable and scalable benchmarks [35]. As shown in Figure 12, there is not much parallelism in non-scalable applications, such as *h2* and *jython*, and performance deteriorated or stagnated as the number of mutators increased. In contrast, for scalable applications, such as *lusearch*, *sunflow* and *xalan*, performance improved with more mutators but became non-scalable as mutator concurrency continued to increase. We had three observations: 1) GC overhead depends on mutator activities. For non-scalable applications, more mutators did not exploit much parallelism, thereby not creating more activities on the heap. GC time was relatively stable with a varying number of mutators. For scalable applications, GC overhead increased with mutator activities as load imbalance could arise. The increased GC time could possibly affect application scalability. For example, the GC time jump in *lusearch* from 4 to 8 mutators was the inflection point of its scalability curve. 2) The optimized JVM significantly reduced GC time in all applications and was not sensitive to mutator activity. Hence, it does not affect application scalability. 3) The improved load balance in GC also helped improve mutator performance as the overall performance gain in all applications was greater than the savings on GC time. An explanation is that good load balance during GC keeps multiple cores active. When the STW pause ends, waking mutator threads will have lower startup latency on the active cores compared to waking up from idle cores in deep power saving states. Further, active cores perform more frequent load balancing and help prevent mutator imbalance.

5.5 Application Results

Big data applications. We used *wordcount*, *pagerank* and *kmeans* from *HiBench* to evaluate the effectiveness of our design in a Spark environment. We set the heap size of Spark executors to 16GB and mutator thread to 16. Figure 13 (a) shows the performance of these applications with three pre-defined data sizes: small, large and huge. Note that *pagerank* with the huge dataset crashed due to out-of-memory errors and was not included in the figure. The results show that the optimized JVM consistently outperformed the vanilla JVM. The performance improvement on individual applications increased as the datasets became larger. However, the overall performance improvement of big data applications was not as great as *DaCapo* or *SPECjvm2008* benchmarks. For example, the greatest performance gain was 15.3% observed in *kmeans* with the huge dataset. The improvement was from reduced GC and mutator time. In *kmeans*, the GC time accounted for 50.3% of the total execution time, about two thirds of which was due to full GC. It has been reported that GC on the old generation is the bottleneck in big data application due to the caching of Resilient Distributed Dataset (RDD) [29]. Thus, scanning such a huge object in the heap is different from other GC tasks. The optimized JVM mainly reduced the time in minor GC for big data applications.

Database applications. Figure 13 (b) and (c) show the latency of read and write requests in *Cassandra* database. *Cassandra* was launched in a JVM with a 8GB heap and the client machine remotely sent requests using 256 threads. The concurrency setting on the client achieved the maximum *Cassandra* throughput. The results show that our optimized JVM was most effective on reducing the tail latency while had marginal improvement on the mean and median latency. It improved the 99^{th} percentile read latency in *Cassandra* by 43%. The improvement on the tail latency led to up to 31% increase on *Cassandra* throughput (not shown in the figure).

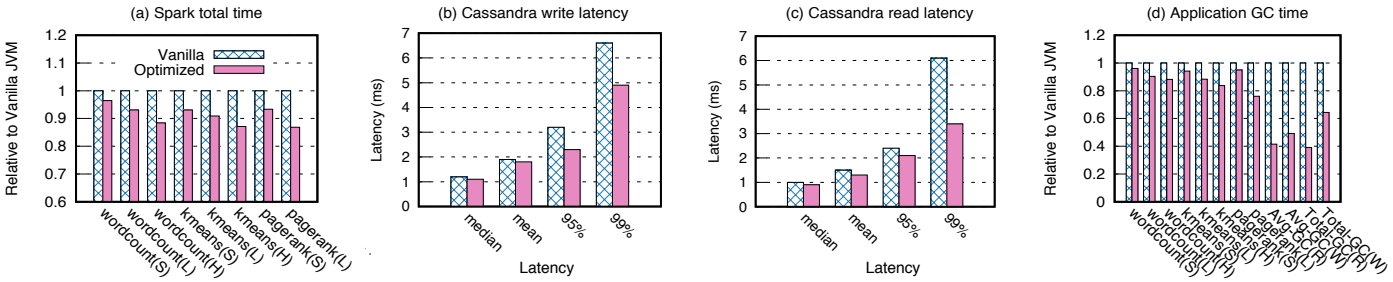

Figure 13: Application performance. ('S', 'L' and 'H' denote the small, large and huge data size, respectively, in HiBench.)

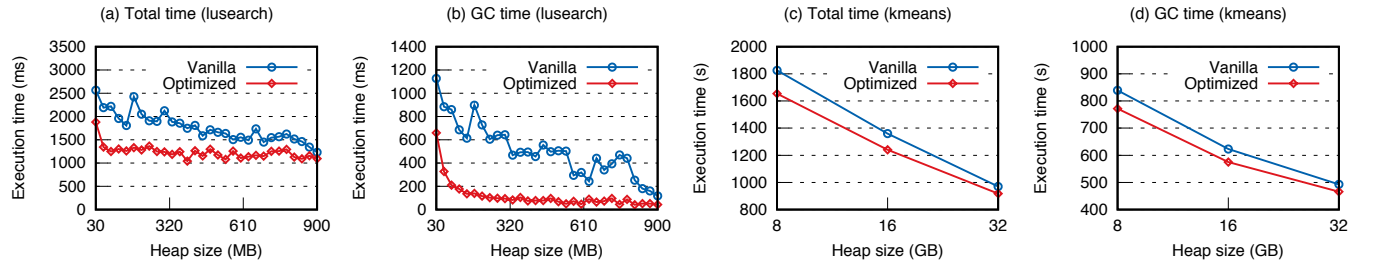

Figure 14: The overall execution time and GC time of *lusearch* and *kmeans* with various heap sizes.

In what follows, we show more detailed statistics on GC time of these applications.

Application GC time. Figure 13 (d) shows the normalized GC time in Spark and *Cassandra* database. As discussed earlier, the saved GC time in the optimized JVM contributed most to overall performance improvement of big data applications. Compared to the small data size, the optimizations achieve more improvement of GC on the large data size. For database operations, both the total and average GC time in *Cassandra* are reduced in the optimized JVM. The optimized GC time was only about half of that in the vanilla GC. The reduction on the STW pause helped rein tail latency.

5.6 Results on Different Heap Sizes

For Java applications, it is important to set an appropriate heap size to prevent too frequent garbage collection while minimizing memory usage. This section studies the performance of the optimized JVM with different heap sizes. The smaller the heap size, more frequently would GC be performed but each GC takes less time as the space to scan is smaller. On the other hand, a larger heap requires GC to scan more space and each GC takes longer. Figure 14 shows the total execution time and GC time of *lusearch* and *kmeans* with different heap sizes. *lusearch* started with a minimum 30MB heap and increased the heap size at a step of 30MB until reaching 900MB. Both JVMs had improving performance as the heap size of *lusearch* increased. The GC time also continued to drop with larger heaps. At the minimum heap size, the performance gain due to our optimizations is narrower than that with larger heap sizes. The reason is that within such a small heap, the overhead of managing multiple GC threads may outweigh the benefit of concurrency. Improved load balance in the optimized JVM can hurt performance due to loss of locality. Nevertheless, the optimized JVM still outperformed the vanilla JVM by a large margin. The vanilla JVM can achieve comparable performance with the optimized JVM only with a much large memory footprint. Figure 14 (c) and (d) show a slightly different trend in *kmeans*. Instead of suffering poor GC performance with the minimum heap size in *lusearch*, the optimized JVM attained the most performance gain over vanilla JVM with 8GB minimum heap and had diminishing gain as heap size increased. As the proportion of the GC time in the total execution time decreases with larger heap size, performance improvement which derives from the optimization on GC time reduction diminishes as the GC time becomes less significant to the total time.

5.7 Results with Multiple Applications

We further evaluated our optimizations in a multi-application environment. First, we ran ten busy loops on ten cores to emulate some background workloads contending for the CPU. When *lusearch* is run along with the interfering workload, the OS load balancer tends to stack the GC threads onto a few cores. Static binding may also place GC threads with the interfering loop on the same core. As Figure 15 (a) and (b) depict, dynamic GC thread balancing was able to reduce the total execution time and GC time of *lusearch* by 49.6% and 77.2% compared to the vanilla JVM. In addition, we evaluated the performance of multiple co-running JVMs. We executed two instances of *lusearch* or two instances of *sunflow* at the same time and the thread settings as well as the heap size were the same as above. Figure 15 (a) and (b) illustrate the improvement of total execution time and GC time of two *lusearch* or *sunflow*. When two JVM applications execute simultaneously, both the total and GC time increase compared to executing them alone in Figure 10. However, our optimizations still benefit both the overall and GC performance. This is due to the GC load balancing and smart stealing that help the applications perform more efficiently under constrained resources.

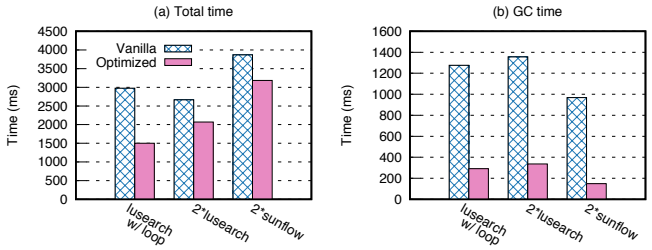

Figure 15: Performance of the total and GC time in a mixed application environment.

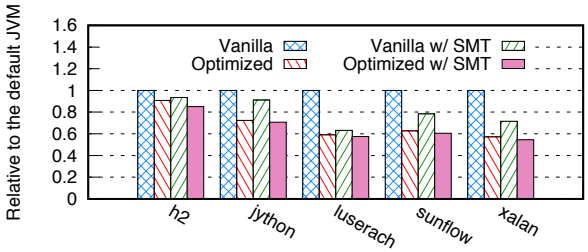

Figure 16: Performance in the vanilla and optimized JVM with or without SMT enabled.

5.8 Discussion

Simultaneous multithreading. While existing studies have reported both constructive and destructive effects of SMT on application performance [15, 19, 26, 36], we found that enabling SMT can mitigate the thread stacking issue. For a fair comparison and to avoid over-threading in GC, we fixed the number of GC threads to 15 to match the number of GC threads automatically determined by HotSpot when SMT was disabled. Figure 16 shows that enabling SMT improved the overall performance. In CFS, the default interval between periodic load balancing is 64*ms* with SMT enabled and 128*ms* with SMT disabled. Thread migration between sibling hyperthreads is considered cheaper than that across physical cores and is thus performed more frequently. In addition, with SMT enabled, cores are less likely to enter a low-power state as activities on either of its two hyperthreads will prevent the core from idling. This will also increase the frequency of `idle_balance` in CFS. Nevertheless, SMT does not eliminate thread stacking and our approach further improves performance via thread affinity and adaptive stealing.

Beyond garbage collection. There exists an inherent tradeoff between optimizations on synchronization and OS scheduling. On the one hand, synchronization optimizations limit the number of concurrent lock contenders to reduce either futile park-unpark activity or CAS contention. On the other hand, OS load balancing is effective only if threads that are to be balanced are visible to the OS scheduler. Blocked threads are not eligible for load balancing as they do not contribute to the load. Besides parallel GC, we also observed a similar thread stacking issue in the `futex-wake` benchmark from `perf` benchmarks. The common issue is that programs with fine-grained blocking synchronization suffer execution serialization because the OS load balancing is ineffective. To address this issue, the OS could choose to balance blocked threads or rely on applications to provide hints on how to distribute threads on cores.

6 RELATED WORK

6.1 Optimizing Garbage Collection

Many studies [3, 6, 12, 23, 35, 43, 44, 49] have demonstrated that it is effective to improve Java application performance by reducing the GC overhead. For instance, Yu et al. [49] discovered unnecessary memory accesses and calculations during the compaction phase of a full GC and proposed an incremental query model for reference calculation to accelerate the GC. To address deficiencies in the existing JVMs, many researchers [10, 18, 23, 39, 43] proposed specialized garbage collectors or re-designed the JVM runtime. Some other works [4, 31] proposed new interfaces to facilitate the interaction between applications and garbage collectors. Such designs require changes to the application source code. In this work, we identified a previous unknown performance issue due to the lack of coordination between parallel GC and the underlying OS. We proposed two optimizations in HotSpot, which are transparent to user programs.

Memory locality and access balance on NUMA multicore machines attracted much attention in recent years. Gidra et al. [8] studied the scalability of throughput-oriented GCs and proposed to map pages and balance GC threads across NUMA nodes. NumaGiC [9] focused on improving memory access locality without degrading the GC parallelism. It avoided costly remote memory accesses and restricts GC threads to collect its own node's memory. While these works employed node affinity to aid NUMA-aware memory management, our work uses thread affinity to avoid harmful interactions between the JVM and the OS scheduler. Sartor et al. [38] found that most benchmarks did not benefit from thread pinning in the Jikes Research Virtual Machine (RVM) [37]. This paper discovered a performance issue due to unfair locking in the HotSpot JVM and Linux CFS.

Work stealing dynamically balances tasks among threads to improve GC efficiency [8, 9, 12, 34, 44]. Gidra et al. [8, 9] proposed to restrict work-stealing to GC threads that run on the same node. Wessam [12] introduced an optimized work-stealing algorithm that uses a single thread to accelerate the termination phase. Wu et al. [44] proposed task-pushing instead of work-stealing to improve stealing efficiency. Qian et al. [34] replaced the `steal_best_of_2` design with a heuristic-based stealing algorithm. A GC thread immediate aborts stealing if failed and only steals from the same victim if stealing was successful. Although this algorithm reduced the number of failed steal attempts, it undermined concurrency during work stealing. In contrast, we proposed to optimize the existing best_of_2 random stealing algorithm and accelerate the termination phase by adaptively altering the termination criteria based on the number of active GC threads.

6.2 Accelerating Java Applications

Existing studies [35, 45, 47] improved Java performance by adopting an efficient scheduler. For instance, Qian et al. [35] found that the FIFO scheduler in Linux achieved higher GC efficiency and mutator performance due to less heap competition and fewer context

switches. A recent work [21] found several bugs in CFS that cores are left idle when there are runnable threads, and proposed fixes. The GC inefficiency identified in this paper is caused by the lack of coordination between the JVM and Linux CFS, and thus the fixes had no effect on the thread stacking issue.

As big data applications are becoming increasingly popular, many researchers start to study optimizations of data-intensive workloads on distributed systems through improved memory management [10, 22] or JVM runtime [18, 23, 25, 29, 30]. Another trend in optimizing Java performance is to coordinate with hardware architectures. Studies [3, 8, 9, 38] explored NUMA-aware designs to improve GC locality and application performance. Maas et al. [24] proposed a hardware-assisted GC which had high throughput and good memory utilization to overcome the STW pause. Hussein et al. [16] proposed a GC-aware governor balance on-chip energy consumption and the performance. These studies are orthogonal to our work.

7 CONCLUSION

In this paper, we identified vulnerabilities in the HotSpot JVM and Parallel Scavenge that can inflict loss of concurrency in parallel GC. We performed an in-depth analysis of the issue and revealed that it resulted from complex interplays among dynamic GC task assignment, unfair `mutex` locking, imperfect OS load balancing and less efficient stealing during the GC. We proposed an effective approach which coordinated the JVM with the Operating System to address GC load imbalance and designed a more efficient work stealing algorithm. Experiment results showed consistent performance improvement compared to the vanilla HotSpot JVM in various types of applications.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their comments on this paper and our shepherd Tim Harris for his suggestions. We also thank Lokesh Gidra for sharing the code of his work. This work was supported in part by U.S. NSF grants CNS-1649502 and IIS-1633753.

REFERENCES

[1] Andrew W Appel. 1989. Simple generational garbage collection and fast allocation. In *Software: Practice and Experience*.
[2] Cassandra. 2008. *Cassandra.* http://cassandra.apache.org/.
[3] Kuo-Yi Chen, J Morris Chang, and Ting-Wei Hou. 2011. Multithreading in Java: Performance and scalability on multicore systems. In *IEEE Transactions on Computers*.
[4] Nachshon Cohen and Erez Petrank. 2015. Data structure aware garbage collector. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*.
[5] DaCapo. 2009. *DaCapo Benchmarks.* http://dacapobench.org/.
[6] Hua Fan, Aditya Ramaraju, Marlon McKenzie, Wojciech Golab, and Bernard Wong. 2015. Understanding the causes of consistency anomalies in Apache Cassandra. In *Proceedings of the VLDB Endowment*.
[7] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2011. Assessing the scalability of garbage collectors on many cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems(PLOS)*.
[8] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2013. A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[9] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: A garbage collector for big data on big NUMA machines. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[10] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS)*.
[11] Hadoop. 2011. *Hadoop.* http://hadoop.apache.org/.
[12] Wessam Hassanein. 2016. Understanding and improving JVM GC work stealing at the data center scale. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*.
[13] Matthew Hertz and Emery D Berger. 2005. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA)*.
[14] Hibench. 2016. *Hibench.* https://github.com/intel-hadoop/HiBench.
[15] Wei Huang, Jiang Lin, Zhao Zhang, and J Morris Chang. 2005. Performance characterization of Java applications on SMT processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
[16] Ahmed Hussein, Antony L Hosking, Mathias Payer, and Christopher A Vick. 2015. Don't race the memory bus: taming the GC leadfoot. In *Proceedings of the International Symposium on Memory Management (ISMM)*.
[17] Kafka. 2011. *Kafka.* https://kafka.apache.org/.
[18] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. 2016. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[19] Jack L Lo, Luiz André Barroso, Susan J Eggers, Kourosh Gharachorloo, Henry M Levy, and Sujay S Parekh. 1998. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*.
[20] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2012. Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (ATC)*.
[21] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*.
[22] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. 2016. Lifetime-Based Memory Management for Distributed Data Processing Systems. In *Proceedings of the VLDB Endowment*.
[23] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[24] Martin Maas, Krste Asanovic, and John Kubiatowicz. 2016. Grail Quest: A New Proposal for Hardware-assisted Garbage Collection. In *6th Workshop on Architectures and Systems for Big Data (ASBD)*.
[25] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. 2015. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS)*.
[26] Harry M Mathis, Alex E Mericas, John D McCalpin, Richard J Eickemeyer, and Steven R Kunkel. 2005. Characterization of simultaneous multithreading (SMT) efficiency in POWER5. In *IBM Journal of Research and Development*.
[27] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*.
[28] José E Moreira, Samuel P Midkiff, Manish Gupta, Peng Wu, George Almasi, and Pedro Artigas. 2002. NINJA: Java for high performance numerical computing. In *Scientific Programming*.
[29] Khanh Nguyen, Lu Fang, Guoqing Harry Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.
[30] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[31] Diogenes Nunez, Samuel Z Guyer, and Emery D Berger. 2016. Prioritized garbage collection: explicit GC support for software caches. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.
[32] Jiannan Ouyang and John R. Lange. 2013. Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*.
[33] PATCH. 2012. *Implement -XX:+BindGCTaskThreadsToCPUs for Linux.* http://mail.openjdk.java.net/pipermail/hotspot-dev/2012-July/006222.html.
[34] Junjie Qian, Witawas Srisa-an, Du Li, Hong Jiang, Sharad Seth, and Yaodong Yang. 2015. SmartStealing: Analysis and Optimization of Work Stealing in Parallel

Garbage Collection for Java VM. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ)*.

[35] Junjie Qian, Witawas Srisa-an, Sharad Seth, Hong Jiang, Du Li, and Pan Yi. 2016. Exploiting FIFO Scheduler to Improve Parallel Garbage Collection Performance. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*.

[36] Yaoping Ruan, Vivek S Pai, Erich Nahum, and John M Tracey. 2005. Evaluating the impact of simultaneous multithreading on network servers using real hardware. In *Proceedings of the ACM SIGMETRICS*.

[37] Jikes RVM. 2012. *Jikes Research Virtual Machine*. http://www.jikesrvm.org/.

[38] Jennfer B Sartor and Lieven Eeckhout. 2012. Exploring multi-threaded Java application performance on multicore hardware. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[39] Rifat Shahriyar, Stephen M Blackburn, and Kathryn S McKinley. 2014. Fast conservative garbage collection. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[40] Spark. 2014. *Spark*. http://spark.apache.org/.

[41] SPECjvm. 2008. *SPECjvm2008 Benchmarks*. https://www.spec.org/jvm2008/.

[42] Guillermo L Taboada, Sabela Ramos, Roberto R Expósito, Juan Touriño, and Ramón Doallo. 2013. Java in the High Performance Computing arena: Research, practice and experience. In *Science of Computer Programming*.

[43] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The continuously concurrent compacting collector. In *Proceedings of the international symposium on Memory management (ISMM)*.

[44] Ming Wu and Xiao-Feng Li. 2007. Task-pushing: a scalable parallel GC marking algorithm without synchronization operations. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*.

[45] Feng Xian, Witawas Srisa-an, and Hong Jiang. 2007. Allocation-phase aware thread scheduling policies to improve garbage collection performance. In *Proceedings of the 6th international symposium on Memory management (ISMM)*.

[46] Feng Xian, Witawas Srisa-an, and Hong Jiang. 2007. Microphase: an approach to proactively invoking garbage collection for improved performance. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[47] Feng Xian, Witawas Srisa-an, and Hong Jiang. 2008. Contention-aware scheduler: unlocking execution parallelism in multithreaded Java programs. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA)*.

[48] Yang Yu, Tianyang Lei, Haibo Chen, and Binyu Zang. 2015. OpenJDK Meets Xeon Phi: A Comprehensive Study of Java HPC on Intel Many-Core Architecture. In *the 44th International Conference on Parallel Processing Workshops (ICPPW)*.

[49] Yang Yu, Tianyang Lei, Weihua Zhang, Haibo Chen, and Binyu Zang. 2016. Performance Analysis and Optimization of Full Garbage Collection in Memory-hungry Environments. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*.