Understanding and improving JVM GC work stealing at the data center scale

Conference Paper · June 2016

DOI: 10.1145/2926697.2926706

Understanding and Improving JVM GC Work Stealing at the Data Center Scale

Wessam Hassanein
Google Inc., Mountain View, CA, USA
wessam@google.com

Abstract

Garbage collection (GC) is a critical part of performance in managed run-time systems such as the OpenJDK Java Virtual Machine (JVM). With a large number of latency sensitive applications written in Java the performance of the JVM is essential. Java application servers run in data centers on a large number of multi-core servers, thus load balancing in multi-threaded GC phases is critical. Dynamic load balancing in the JVM GC is achieved through work stealing, a well known and effective method to balance tasks across threads. This paper analyzes the JVM work stealing behaviour, and introduces a novel work stealing technique that improves performance, GC CPU utilization, scalability, and reduces the cost of jobs running on Google's data-centers. We analyze both the DaCapo benchmark suite as well as Google's data-center jobs. Our results show that the Gmail front-end server shows a 15-20% GC CPU reduction, and a 5% CPU performance improvement. Our analysis of a sample of ~59K jobs shows that GC CPU utilization improves by 38% geomean, 12% weighted geomean. GC pause time improves by 16% geomean, 20% weighted geomean. Full GC pause time improves by 34% geomean, 12% weighted geomean.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors—memory management, optimization, run-time environments

General Terms Algorithms, Languages, Performance

Keywords *work stealing, garbage collection, dynamic load balancing*

ISMM'16, June 14, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4317-6/16/06...$15.00
http://dx.doi.org/10.1145/2926697.2926706

1. Introduction

Today Java is one of the most popular languages for application development. Large scale real-time data-intensive Java applications run in large data centers taking advantage of heterogeneous multi-core hardware. Java applications include Google Web Search, Gmail, Docs, social networking, etc. These latency critical (LC) applications [21] process large amounts of data and have tight response time requirements. These requirements are set by service level agreements (SLAs) [21] in data centers. Hundreds to thousands of parallel threads are used by applications. Thus real time performance of the JVM is imperative.

A data-center houses a *cluster* [21] of machines. Borg [21] is Google's cluster manager which runs hundreds of thousands of applications. An application (submitted as a Borg *Job*) usually consists of hundreds of *tasks*. Each job runs in one Borg *cell*, a set of machines managed as a unit. A *task* consists of the application binary, associated data, and a Borg configuration file that specifies application requirements such as processor architecture, number of cores, memory, disk space, as well as machine scheduling requirements, such as only a single task to run on the machine. Borg handles task scheduling, reserves resources, and uses resource containers to isolate different tasks running on the same machine. Borg also monitors tasks, can change resources reserved, and can kill tasks that use more resources than requested. Hence seemingly unhealthy jobs due to large GC pause times can get killed.

A major component of JVM performance is Garbage Collection. GC performs automatic memory management, one of the main advantages of managed languages such as Java, where developers need not worry about de-allocating memory but can rely on GC. GC contains multiple stop-the-world phases where all application threads are halted and only GC threads are executing. At scale, GC CPU utilization translates into real resource usage and power consumption. Thus reducing GC pause times, GC CPU utilization, and improving scalability for such phases is critical. Dynamic load balancing in the OpenJDK JVM GC is achieved through work stealing [2,3,4,5,6,8,16,18,26,27,28,29], a well known and effective method to balance *gc-tasks* across threads. In this paper we show wasted CPU utilization in the current JVM GC work-stealing mechanism and present a new work-stealing technique that considerably reduces *GC CPU utilization* (percentage of CPU time spent in GC for that workload) and improves scalability and GC performance.

In this paper we characterize the behaviour of the OpenJDK JVM GC in Google's Borg managed data-centers showing the effects of the newly proposed work-stealing protocol on large scale data center deployment within Google. We use Google Wide Profiling (*GWP)* [24, 25] to gather hardware-counter performance data as well as job profiling data. We present both data-center scale data as well as isolated machine results that reduce both noise and result variations. We also present a study of the Gmail front-end server application (a highly scalable application) showing the effects of the newly proposed work-stealing technique.

This paper is organized as follows: Section 2 describes Google's OpenJDK-based JVM GC. In Section 3, we present the details of a new work-stealing technique namely; Optimized Work Stealing Threads (OWST). Sections 4 and 5 discuss the experimental methodology and results, respectively. Section 6 presents related work. Finally, conclusions are provided in Section 7.

2. Background

One of the main features of managed runtime systems is automatic memory management through Garbage collection (GC). The OpenJDK JVM has several types of GCs including; Serial, Parallel Scavenge (PS), Concurrent Mark-Sweep (CMS), and G1. Due to performance reasons on multi-core hardware the parallel collectors have been widely preferred over the serial collector and are studied in most previous work [5,12,13,14,15,19,20,23]. The G1 collector has not been widely accepted for use in our applications as of yet due its lagging performance to CMS. However, future improvements to the G1 collector are expected in future JDK versions (e.g. JDK9) and hence could be potential future work. The proposed work-stealing algorithm improves both CMS and PS garbage collectors. This paper targets the CMS collector which is widely used and is the default collector at Google for JDK8. CMS features both concurrent as well as parallel multithreaded garbage collection phases. CMS is introduced in the following subsection.

2.1 The OpenJDK JVM CMS Garbage Collector

The CMS collector is a mostly-concurrent semi-space generational collector [23,31] (based on the hypothesis that most objects die young) where the heap is divided into the young generation (*YoungGen*) and the old generation (*OldGen*). The young generation is further divided into the Eden space, and two survivor spaces known as S0 and S1. Within our JVM garbage collection, 3 separate collectors are used; 1- *ParNew*: YoungGen stop-the-world copying collector. 2- *CMS*: Concurrent + stop-the-world phases, mark and sweep collector. 3- *Full GC*: OldGen stop-the-world parallel mark-sweep-compact collector. This is a parallel version of the OpenJDK full GC collector developed within Google. The default full GC collector within the OpenJDK is a serial collector.

Both concurrent and parallel GC phases are multithreaded. Concurrent phases execute while the application is running, while stop-the-world phases cause all application threads to be halted. Hence stop-the-world GC pauses are critical for application performance and response time.

2.2 The CMS GC Parallel Thread Task Queues

The OpenJDK JVM uses a GenericTaskQueue which implements an ABP, Aurora-Blumofe-Plaxton [1], double-ended-queue (deque) for use in work stealing. Queue operations are non-blocking. A queue owner thread performs push and pop_local operations on one end of the queue, while other threads may steal work using the pop_global method. The main difference to the original ABP algorithm is that the OpenJDK JVM implementation allows wrap-around at the end of its allocated storage, which is an array. [2] provides the correctness proof and an implementation for weakly ordered memory models including (pseudo-) code containing memory barriers for a Chase-Lev deque [17,18]. Chase-Lev is similar to ABP, with the main difference that it allows resizing of the underlying storage. pop_local_slow is used by the owning thread when it is trying to get the last task in the queue. It will compete with pop_global that is used by other threads. Only a single task gets stolen at a time from the deque.

2.3 The CMS GC Parallel Work-Gang

During JVM initialization the VM (Virtual Machine) thread initiates two sets of work-gangs; the *Parallel-GC work-gang* and the *Concurrent-GC work-gang*. Each work-gang contains a number of threads that will be used during the different phases of GC. The number of threads and priority is set at this initial stage, thus determining the max number of threads that can be used. A subset of the work-gang can be used through *FlexibleWorkGangs*. In this work we are primarily concerned with the Parallel-GC work-gang. After each GC thread is initialized, it enters an infinite double loop where a monitor is used to control the threads.

The VM thread does initial work distribution and setup and calls *WorkGang::run_task* for the GC worker threads to execute the task. This is done through a *monitor->notify_all* call. The VM thread then waits until the number of *finished_workers* equals the *no_of_parallel_workers* through a *monitor->wait* call. Once all workers have finished their work the task is complete and the VM thread resumes execution. This is repeated per task for the different phases of GC.

2.4 Dynamic Load Balancing and Work Stealing in GC

Dynamic load balancing (DLB) is a crucial part of optimizing the performance of parallel GC threads. DLB makes sure that no thread is overloaded while other threads are idle. Hence, it redistributes the tasks between threads dynamically. Work stealing is a well known DLB approach [2,3,4,5,6,8,16,18] that is used in the OpenJDK JVM.

Figure 1 shows a stop-the-world GC pause. In a stop-the-world GC pause, all mutator (application) threads are halted and the VM thread takes over. The VM thread distributes the tasks across the GC parallel threads in the parallel-GC work-gang. The VM thread issues the run command to the GC parallel threads and suspends itself, waiting for the GC threads to complete their work. When all GC threads enter the parallel GC thread termination phase, this signals the end of the GC work and the VM thread regains control. The VM thread then awakens the mutator threads and the application continues its execution.

JVM GC work stealing is achieved as follows; After a GC thread finishes all its work, it checks to see whether it can steal work from other threads' task queues. If it can, it steals one task at a time and executes it. The task stealing cycle continues until no further tasks are available to steal in all the task queues. When a thread can no longer steal any work it enters the parallel GC thread termination phase as shown in Figure 1.

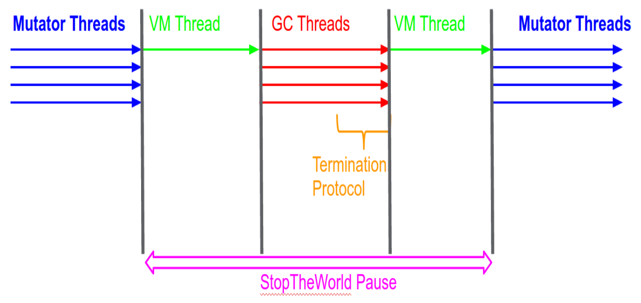

Figure 1. Stop-the-world GC phase showing the parallel GC thread termination protocol phase.

The OpenJDK JVM parallel-GC thread termination phase consists of 3 states; 1- CPU spinning in a spin loop with an exponentially increasing number of iterations (up to 4K spin loop iterations). Ten state 1 invocations before going to the next yield state. 2- A yield state where the GC thread does an OS yield call. By default, 5K yields before going to the next sleep state. 3- A sleep 1ms state. A GC thread entering the termination phase goes through a loop of all 3 states in-order checking whether work exists to steal at each state transition. If work exists to steal, a GC thread exits the termination phase and continues with its work-stealing and execution. At each state transition the number of GC threads in the parallel-GC thread termination phase is checked. If all threads are offering termination, then all threads exit and the termination phase ends. All GC threads return to the parallel-GC threads work-gang and the VM thread resumes.

The OpenJDK JVM work stealing algorithm used to decide which task queue to steal from is a maximum of two random selections. This is repeated 2n times, where n is the number of task queues. This algorithm performs quite well according to our experiments and outperforms both an overall maximum algorithm as well as a random algorithm. These results agree with previous publications [9].

We have identified parallel task termination as a cause for wasted GC CPU cycles through a perf hardware events cycle-accounting analysis on the DaCapo benchmark suite described in Section 4.1.1 and through a Google-wide GWP analysis as described in Section 4.1.2.

3. Optimized Work-Stealing Threads (OWST)

In this section we present a new work stealing technique where the parallel-GC thread termination protocol reduces CPU spinning considerably, is more scalable, improves GC pause times, and reduces GC thread contention. We refer to our new technique as Optimized Work-Stealing Threads (OWST).

In OWST only a single GC thread is used as the master thread (*spin-master*). The *spin-master* thread is identified as the first GC thread that enters the parallel-GC thread termination phase if no other GC thread has already been identified as a *spin-master.* All other GC threads that enter the parallel GC thread termination phase enter a wait state. The *spin-master* thread is the only thread that spins and the only thread that checks for work to steal. This is achieved by checking all other thread queues. If work exists to steal, the *spin-master* thread calculates the number of GC threads needed for this work based on the number of tasks available to steal, and wakes up the specified number of threads -1. The *spin-master* thread then exits the parallel-GC thread termination phase relinquishing its *spin-master* status to execute as a worker thread its share of work stealing. All threads that have been woken up from the previous wait state will steal and execute the stolen tasks until no further work can be stolen. Then the parallel-GC thread termination phase will start again with a new *spin-master.* Upon the final GC thread entering the parallel-GC thread termination phase, all waiting threads are woken up and exit the parallel-GC thread termination phase, thus ending execution of this phase.

Although threads can race to acquire the *spin-master* status, which thread wins doesn't make a difference. Only the first thread that acquires the *spin-master* lock becomes the *spin-master,* all other threads will be wait-type threads. There are no race conditions to enter sleep, or to wake up sleepers.

Threads that go into the wait state periodically wake up and check whether they should continue into another wait state or not. If tasks are available to steal they should exit the wait state and steal them.

OWST uses the queue length as a good predictor of work to be done because the queue holds work at the granularity of a task which is the granularity at which work is stolen in the OpenJDK JVM. Tasks that represent large sub-graphs are dynamically popped-off the queue by the owning thread, divided into their subtasks, and pushed back into the queue as tasks that can be potentially stolen by other threads. Thus the number of tasks that can be stolen is represented by the overall queue length of all threads which is used by OWST and dynamically represents the work available.

In the OpenJDK JVM a single JVM main thread is used to control all GC threads. Similarly, in OWST a single GC thread is used to wake up N GC threads through the same JVM monitor *wait* and *notify* calls. Only a single thread can acquire the monitor lock at a time to issue these calls. Thus having multiple threads attempt to acquire the monitor lock results in thread contention and performance degradation. Therefore, having a single thread issue the notify calls does not reduce the scalability of the OpenJDK JVM. If the number of tasks to be stolen is larger than the number of waiting threads, then a *notify_all* call is used to wakeup all waiting threads.

OWST reduces CPU spinning considerably as well as thread contention as only a single thread is spinning some of the time and only a single thread is checking for available work to steal. Moreover, unlike the original work stealing technique where all threads are spinning, contending for locks, and checking for work to steal, OWST is a scalable mechanism that eliminates contention on the shared resources. OWST dynamically selects the number of threads needed for the work stealing tasks.

We have experimented with a 1ns sleep only solution as well as a reduction in the number of parallel GC threads remaining in work-stealing. However, our current solution showed better performance, scalability, and dynamic load balancing.

4. Experimental Methodology

4.1 DaCapo Benchmark Suite on Isolated Lab Machines

We use isolated Intel Xeon SandyBridge machines to reduce noise and variations associated with production machines and achieve accurate 1:1 comparison. Our SandyBridge machines are dual-socket, 8 cores each, HT with 32 threads total, with 2MB L2 cache and 20MB L3 cache. Our JVM is based on the OpenJDK JVM with some modifications that improve GC performance such as the parallelization of the full GC phase. We run the DaCapo 2009 Benchmark Suite [22]. To reduce variations and noise we run each benchmark 30 runs per configuration and take the average. Each run containing 9 warm-up runs + 1 measurement run. To test scalability, we measure both 8 parallel GC threads and 23 parallel GC threads configurations. We choose the max heap size per benchmark based on a 2.5x heap multiplier of the GC thrashing heap size. Heap sizes are shown in Table 1.

Table 1. DaCapo 2009 Benchmark Suite max heap size configuration

Benchmark	Heap Size (MB)
avrora	4
luindex	10
lusearch	20
xalan	44
fop	54
jython	54
tomcat	54
sunflow	64
pmd	74
tradesoap	134
tradebeans	140
eclipse	172
h2	574

4.2 Data-Center Scale Data Collection GWP

We use the Google Wide Profiling (GWP) [24, 25] infrastructure to collect hardware counter performance data as well as profiling data. GWP is a low-overhead continuous profiling infrastructure that collects random-sampling based performance data. Results presented in this paper use the Google JVM which is based on the OpenJDK JDK8.

To do an A/B comparison of GWP data from comparable same jobs we compare jobs with the following factors constant; job, program name, user, last change ID, hardware platform, heap size (YoungGen and OldGen), and memory heap committed bytes. We average values from the same jobs and compare two versions of the OpenJDK JVM before OWST enabling and after. We show the geomean across different jobs as well as the weighted geomean where values are weighted by the number of jobs.

5. Results

5.1 DaCapo Benchmark Suite Analysis

In this section we describe the results of DaCapo benchmark suite using the infrastructure described in Section 4.2. The DaCapo benchmark suite gives a wide range of application behaviour resulting from its different applications and hence is useful in understanding the OpenJDK JVM GC behaviour. Our experimental setup described in Section 4.2 reduces experimental noise considerably, thus isolating the effects of changes on the OpenJDK JVM GC. Figures 2 and 3 show the number of work stealing spin invocations for 23 parallel GC threads (23T) and 8 parallel GC threads (8T) respectively. We choose 23T since it is the default OpenJDK parallel GC thread configuration for a 32 thread SandyBridge machine used in our experiments as described in Section 4.1.1. We choose 8T since it is the experimental optimal parallel GC thread configuration based on our performance results for the DaCapo benchmark suite using the OpenJDK JVM. Baseline runs use the default OpenJDK JVM work stealing technique described in Section 2.4. Our results show that OWST reduces the number of spin invocations considerably by up to 97%, 92% on average (23T) and up to 86%, 79% on average (8T). This results in considerable savings in CPU utilization, an important metric at the data-center scale as it translates into real resources usage and power consumption as will be shown in Section 5.1.3.

In Figure 4 we analyze the scalability of OWST compared to the baseline JVM work-stealing. Scalability is a major factor at the data-center scale as application sizes keep increasing as well as memory consumption. Hence, scalable, more efficient memory management is crucial. Our analysis shows that as the number of threads increase from 8 to 23 threads most DaCapo benchmarks suffer an increase in the number of CPU spin invocations per thread. This is due to the contention between threads. However, as we compare the performance of OWST as we scale from 8 to 23 threads we can see that the number of CPU spin invocations per thread in all DaCapo benchmarks decreases indicating a scalable solution. This is due to the replacement of an all thread spin non-scalable work-stealing with a single thread spin OWST scalable technique.

Figures 5 and 6 show the YoungGen and OldGen GC pause times respectively. Both YoungGen and OldGen GC use OWST parallel GC threads and are StopTheWorld phases. Hence, the application is stopped while parallel GC threads are used to collect the heap. OWST reduces YoungGen GC pause time by up to 13% (23T) and (10%) 8T. OWST reduces OldGen GC pause time by up to 45% (23T) and 5% (8T). These GC pause time improvements are a result of the reduction of parallel GC thread termination cycles as well as the improvement of work-stealing task execution by the GC threads. In OWST useless threads no longer try to steal tasks from other useful threads or from each other. OWST reduces the contention on monitors, locks as well as shared hardware resources between the parallel GC threads. This is important as useless threads have no value (no work to execute) and only result in the degradation of the performance of the OpenJDK JVM GC in addition to the wasted cycles both in CPU utilization as well as power consumption. The overall reduction in GC pause times in OWST improves performance by up to 7% (23T and 8T) on the DaCapo benchmark suite.

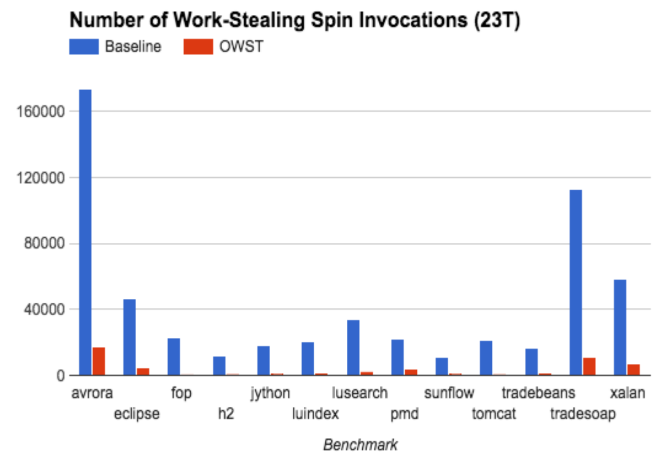

Figure 2. Number of work-stealing CPU spin invocations using 23 parallel GC threads (23T) on the DaCapo Benchmark Suite

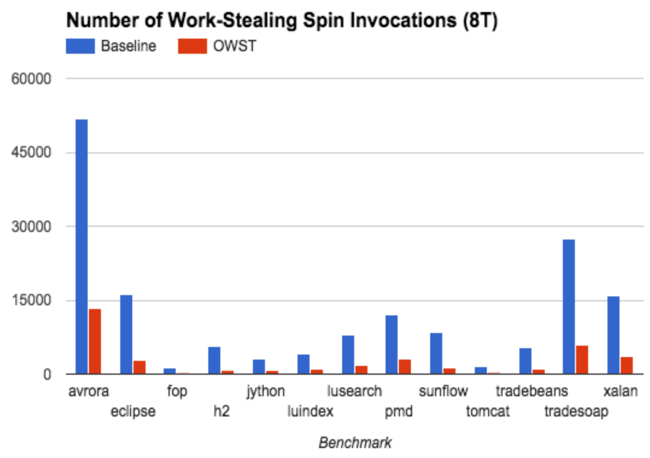

Figure 3. Number of work-stealing CPU spin invocations using 8 parallel GC threads (8T) on the DaCapo Benchmark Suite

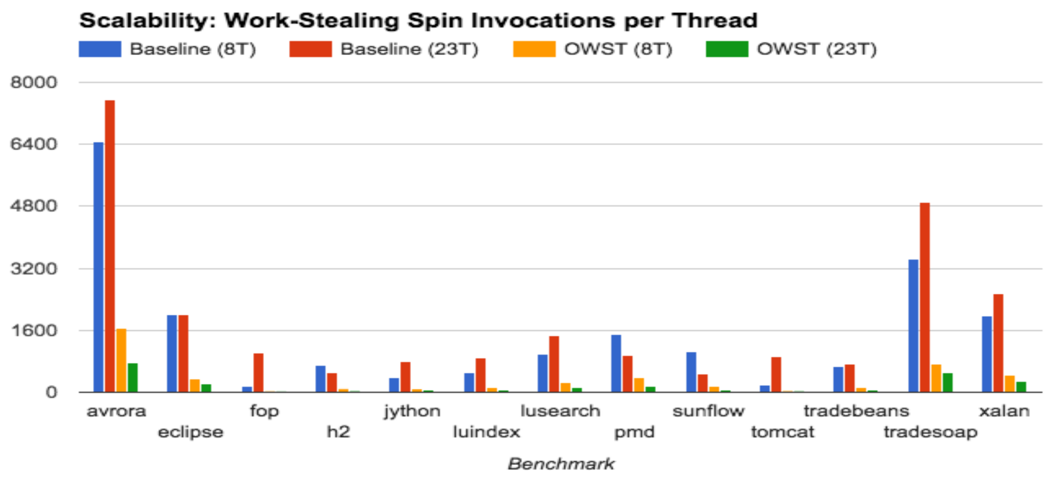

Figure 4. Scalability Analysis. Number of work-stealing CPU spin invocations per thread on the DaCapo Benchmark Suite

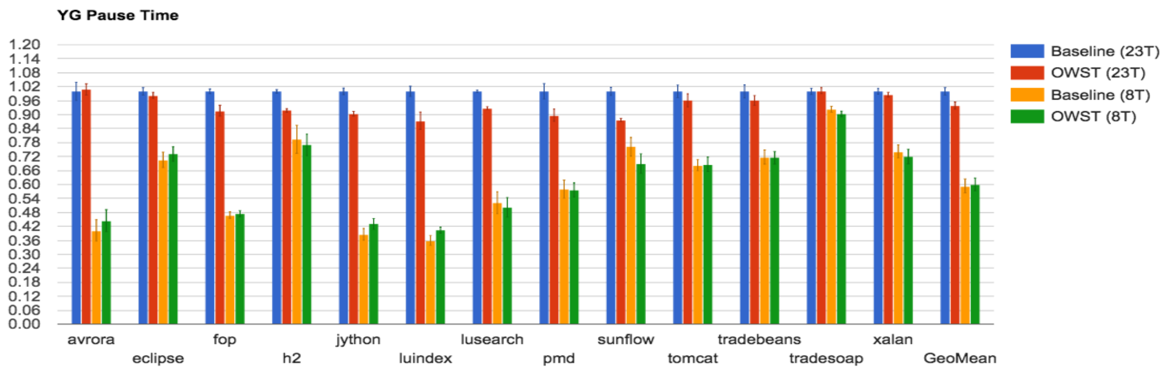

Figure 5. YoungGen GC pause time

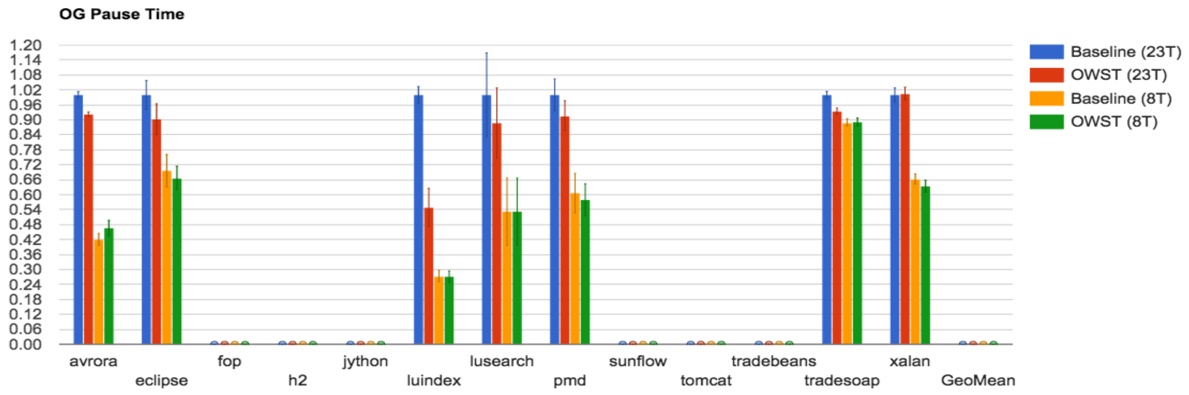

Figure 6. OldGen GC pause time. In some cases, there are no OG GCs.

5.2 Gmail Front-End Server

Gmail is a latency critical application that runs a large number of jobs within Google data centers. The performance of Gmail scales well with JVM GC threads and hence it uses 23 parallel GC threads on an Intel SandyBridge machine. With OWST Gmail shows a 15-20% GC CPU reduction as shown in Figure 7 and a 5% CPU performance improvement as shown in Figure 8. OWST also resulted in a 0.2% memory usage reduction. Results were measured using a control (baseline) and test jobs chosen in the same cell and with the same characteristics. OWST has been enabled by Gmail in production.

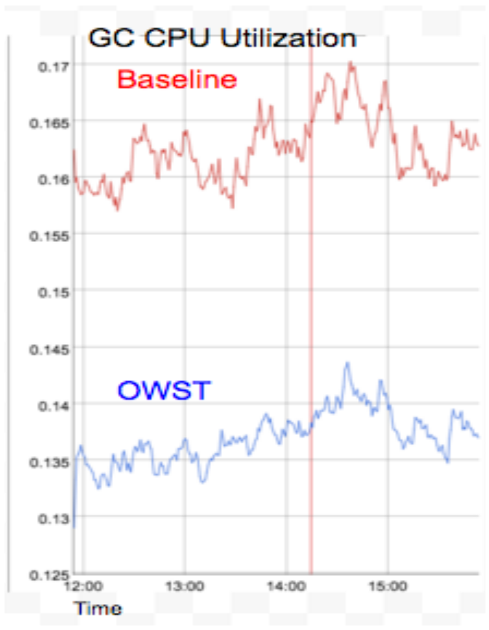

Figure 7. Gmail GC CPU Utilization

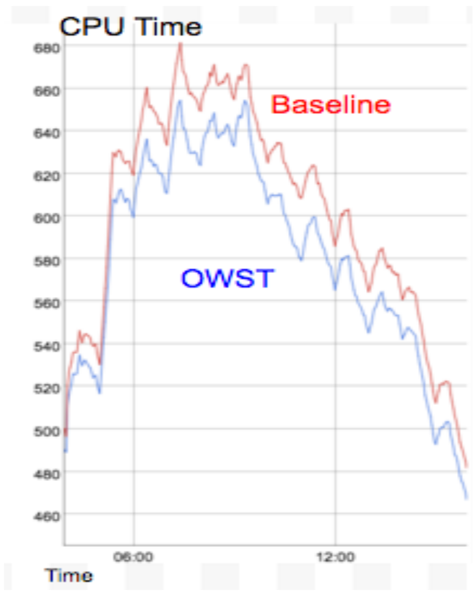

Figure 8. Gmail CPU Time

5.3 Data-Center Scale Results

We have rolled out OWST to production as a default enabled in the Google JDK8 JVM. Initial GWP results have shown that the wasted CPU spin cycles observed in the GC parallel termination phase with our original JVM has been reduced by over 50% by OWST. As the OWST is further adapted in Google applications we expect the wasted CPU spin cycles to go down further.

In this section we analyze the results of a set of ~59K production jobs we gathered through GWP. The results compare two versions of the OpenJDK JVM without (prior to enabling OWST) and with OWST. We use the number of instances of a job as the weight in our weighted-geomean calculations.

We calculate the GC CPU utilization ratio for each job (proportion of total CPU cycles spent in GC) as shown in equation (1). The improvement in GC CPU utilization ratio is calculated by comparing without (Baseline) vs with OWST as shown in equation (2). We calculate GC pause time per GC event as shown in equation (3).

Our results show that overall GC CPU utilization improves by 38% geomean, 12% weighted geomean. GC pause time per GC improves by 16% geomean, 20% weighted geomean. Full GC pause time per GC improves by 34% geomean, 12% weighted geomean.

- Job GC CPU utilization ratio = Avg over all instances of Job (GC time / CPU time) *(1)*

- GC CPU utilization ratio improvement = (Baseline Job GC CPU utilization ratio) / (OWST Job GC CPU utilization ratio) *(2)*

- GC CPU utilization ratio improvement values above 1 indicate a reduction in the CPU time consumed by GC.

- GC time per GC = GC time / GC count. *(3)*

where GC count = the number of GC occurrences.

Figures 9, 10, and 11 show that the Geomean improves (whether weighted by the number of jobs or not) across different applications for GC CPU utilization, GC time, and Full GC time respectively. Figure 9 shows the GC CPU utilization ratio improvements in top application performers from our ~59K job sample. Our top performers contribute ~11K jobs. Figures 10 and 11 show the corresponding improvements in GC pause time and full GC pause time. Weighted geomean GC CPU utilization ratio has improved by up to 3.9x, while weighted geomeans of GC time ratio and full GC time ratio improved by up to 2.05x and 1.34x respectively.

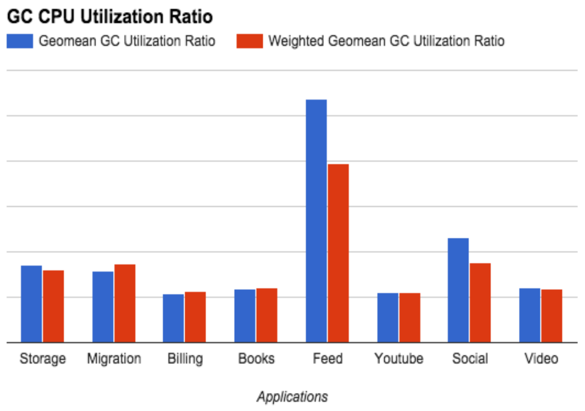

Figure 9. GC CPU utilization ratio of top OWST job performers.

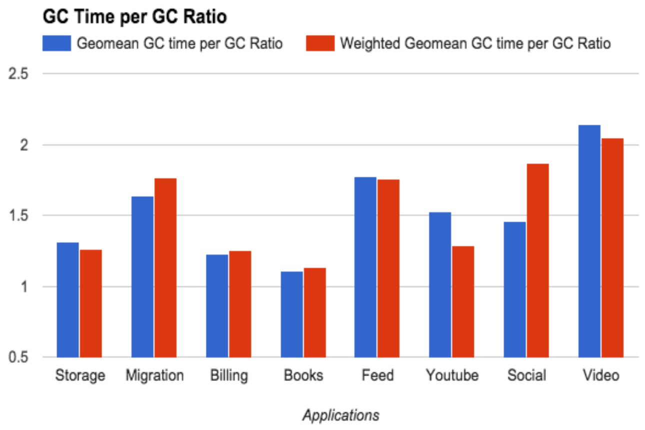

Figure 10. GC time per GC ratio of top OWST job performers.

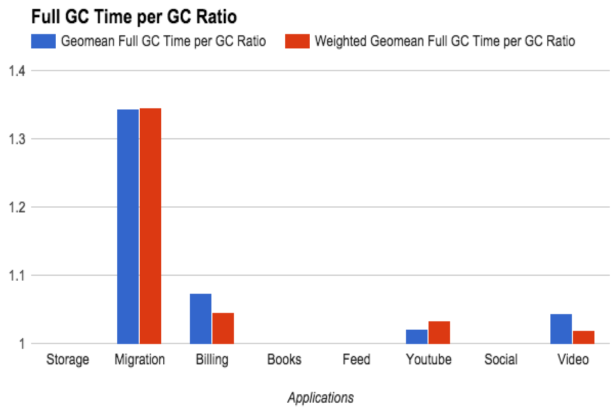

Figure 11. Full GC time per GC ratio of top OWST job performers. In some cases, there are no full GCs.

6. Related Work

Several GC thread load-balancing mechanisms have been proposed in the literature. Pool-sharing [13, 14] uses a global task pool for work-stealing. GC threads that have completed their assigned work can steal tasks from the global task pool. This technique however requires synchronization of the shared task pool and thus results in synchronization overhead and non-optimal performance. In [30] work packets are used instead of mark stacks to distribute objects evenly between threads through a pool where a thread doesn't keep the work it generates.

Task-pushing [11] attempts to eliminate the synchronization overhead by allowing GC threads to add tasks to other GC threads' task Queues (taskQs). In order to achieve this a taskQ for each GC thread is created in each GC thread. Thus the number of taskQs created is n^2 in addition to a private marking stack per GC thread. Synchronization is needed for the queue operation into another GC thread's taskQ. A GC thread is arbitrarily designated to be the termination detecting thread. GC threads have to finish the work on their marking stacks as well as all their taskQs from all other threads. To improve task selection for sharing tasks, marking stacks are allowed to drip from the bottom of the marking task if other threads are free and hence uses the same technique as work-stealing [1]. This old-task dripping achieves the best performance.

Work-Stealing (Task-polling) [5,9,14,15] is the technique currently used in the OpenJDK JVM. GC threads that have completed their assigned task check whether any other GC thread has a task for them to steal, if so, it steals it. Otherwise, the thread spins/yields/sleeps waiting for tasks to be created or the GC phase to end. Only a single task is stolen at a time and hence synchronization only happens on the last task in the taskQ. In [14, 5] stealing half of the tasks in another GC's thread showed good scalability. In [14] each GC thread has a localQ and an auxiliaryQ where it keeps tasks that another thread can steal. Periodically each thread checks its auxiliaryQ and if empty it moves half of its tasks into the empty Q. They proposed a try-lock-then-steal synchronization. In [15] improved over the previous design in [14] by adding a non-blocking double-ended queue. JVM already uses a non-blocking deque with sync only on the last element popQ. Implementing the steal-half technique in the OpenJDK JVM has been shown by [9] to not give performance improvements. Using different stealing algorithms than steal best of 2 random selections currently used by the OpenJDK JVM (e.g. best of all or random selection) has also been shown to not give performance improvements in [9] and agrees with our results.

In [29] Balanced Work Stealing (BWS) a linux OS scheduling algorithm that uses work-stealing is proposed. BWS targets multi-applications running on the OS and hence suffers from different system characteristics and problems. BWS has no spinning as it is a yield/sleep OS scheduling algorithm and threads belong to different applications. This is in contrast to OWST a JVM GC algorithm where spinning is a major component resolved by the OWST algorithm and consumes a large amount of CPU cycles. A yield call as used in BWS will immediately return (effectively continues running/spinning) if there is an available CPU as a yield simply adds the thread to the end of the queue and hence does not resolve the CPU spinning issues that OWST resolves. The OpenJDK JVM work-stealing algorithm is different from the ABP algorithm given in the BWS paper [29], as no yielding occurs before work-stealing. BWS randomly selects a single worker to steal from. In contrast, OWST uses a max of 2 random selections algorithm repeatedly until it finds a task to steal or 2n times, where n is the number of task queues. OWST work-stealing algorithm performs better than just a random selection in our experiments as well as in previous studies [9] as it is better able to find a task to steal if one exists in any thread. BWS relies on each thread using 2 counters to guess how it should behave (each thread has to randomly select a victim and decide whether to yield/sleep). Each thread can wake-up 2 other worker threads and a watchdog thread uses the same algorithm but doesn't sleep. OWST does not use yield or sleep but uses a wait/notify algorithm that is controlled by a single master thread that is aware of how many tasks exist in all queues and is able to notify the correct needed number of waiting threads. Thus OWST does a better decision based on the overall status of all threads' task queues and notifies exactly the number of threads needed.

Mark-Sharing [12] relies on both task-stealing and task-releasing. [12] uses transactional memory (HTM), is simulation-based using the Simics simulator, and uses an in-order microprocessor. Each thread maintains a local taskQ and a globally accessible auxiliaryQ. Each GC thread steals from its own auxiliaryQ first then from any global auxiliaryQ. Task-release occurs when a thread traverses a node with degree > threshold. In this operation the tasks are added to the auxiliaryQ and not the localQ. A selection manager is used to minimize contention and keep task information. It shows which auxiliaryQs have tasks and which to avoid (least recently retained tasks). Thread termination uses distributed local flags and a single global flag. One thread (the selection thread) determines termination only.

Work-sharing [10] uses the address of a word in the mark-bitmap as the key to stripe work among parallel threads. This reduces contention during the update of the mark-bit-map. [16] shows locality guided work-stealing. Message-Passing work-stealing [4] has proposed removing all concurrent deques and replacing them with totally private deques and using message passing for work stealing whether sender or receiver initiated.

7. Conclusion

This paper presents a novel JVM GC work-stealing technique that reduces GC CPU utilization, GC pause time, and improves GC scalability. At scale, GC CPU utilization translates into real resource usage and power consumption. GC CPU spinning in the OpenJDK JVM GC work-stealing parallel task termination is reduced by up to 86% (8T) and 97% (23T) on DaCapo benchmarks. YoungGen GC pause time is reduced by up to 13% (23T) and (10%) 8T. OldGen GC pause time is reduced by up to 45% (23T) and 5% (8T). Performance improved by up to 7% (23T and 8T) on DaCapo benchmarks. A study of Gmail front-end server shows a 15-20% GC CPU reduction, a 5% CPU performance improvement. We further analyze the effects of the presented GC work-stealing technique on the Google data-center production jobs by enabling OWST and rolling out to production in the Google JVM. We use GWP to profile and measure hardware counter performance of data center Jobs. Our results show that GC CPU utilization improves by 38% geomean, 12% weighted geomean. GC pause time improves by 16% geomean, 20% weighted geomean. Full GC pause time improves by 34% geomean, 12% weighted geomean. These improvements translate into real resource savings in CPU and power consumption in the Google data centers.

References

[1] Arora, N. S., Blumofe, R. D., and Plaxton, C. G. Thread scheduling for multiprogrammed multiprocessors. Theory of Computing Systems 34, 2 (2001), 115-144.

[2] Le, N. M., Pop, A., Cohen A., and Nardell, F. Z. Correct and efficient work-stealing for weak memory models. Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP 2013), 69-80.

[3] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, pages 53:1–53:11. ACM, 2009.

[4] Umut A. Acar, Arthur Chargueraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP 2013).

[5] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In PODC, pages 280–289, 2002.

[6] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 45–54, 2009.

[7] Bratin Saha, Ali-Reza Adl-Tabatabai, Anwar Ghuloum, Mohan Rajagopalan, Richard L. Hudson, Leaf Petersen, Vijay Menon, Brian Murphy, Tatiana Shpeisman, Jesse Fang, Eric Sprangle, Anwar Rohillah, and Doug Carmean. Enabling scalability and performance in a large scale chip multiprocessor environment. Technical Report. Intel Corp., 2006.

[8] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A tighter analysis of work stealing. In Algorithms and Computation - 21st International Symposium, ISAAC 2010, Proceedings, Part II, volume 6507 of Lecture Notes in Computer Science, pages 291–302. Springer, 2010.

[9] Helin, Eric. Improving load balancing during the marking phase of garbage collection. 2012

[10] Iyengar, B., Gehringer, E., Wolf, M., & Manivannan, K. (2013). Scalable concurrent and parallel mark. ACM SIGPLAN Notices, 47(11), 61-72.

[11] Wu, M., & Li, X. (2007). Task-pushing: a scalable parallel GC marking algorithm without synchronization operations. Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. IEEE.

[12] Park, H., Lee, C., Kim, S. H., Ro, W. W., & Gaudiot, J. (2013). Mark-Sharing: A Parallel Garbage Collection Algorithm for Low Synchronization Overhead. Parallel and Distributed Systems (ICPADS), 2013 International Conference on. IEEE.

[13] Imai, A., & Tick, E. (1993). Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. Parallel and Distributed Systems, IEEE Transactions on, 4(9), 1030-1040.

[14] Endo, T., Taura, K., & Yonezawa, A. (1997). A scalable mark-sweep garbage collector on large-scale shared-memory machines. Supercomputing, ACM/IEEE 1997 Conference. IEEE.

[15] Flood, C. H., Detlefs, D., Shavit, N., & Zhang, X. Parallel Garbage Collection for Shared Memory Multiprocessors. Java Virtual Machine Research and Technology Symposium. (2001)

[16] Acar, U. A., Blelloch, G. E., & Blumofe, R. D. The data locality of work stealing. Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures. ACM. (2000).

[17] Chase, D., & Lev, Y. Dynamic circular work-stealing deque. Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures. ACM. (2005).

[18] Hendler, D., Lev, Y., Moir, M., & Shavit, N. A dynamic-sized nonblocking work stealing deque. (2005).

[19] Gidra, L., Thomas, G., Sopena, J., & Shapiro, M. A study of the scalability of stop-the-world garbage collectors on multicores. ACM SIGPLAN Notices (2013), 48(4), 229-240.

[20] Gidra, L., Thomas, G., Sopena, J., & Shapiro, M. Assessing the scalability of garbage collectors on many cores. Proceedings of the 6th Workshop on Programming Languages and Operating Systems. ACM. (2011).

[21] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, John Wilkes. (2015). Large-scale cluster management at Google with Borg. Proceedings of the European Conference on Computer Systems (EuroSys), ACM, Bordeaux, France (2015).

[22] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., et al. The DaCapo benchmarks: Java benchmarking development and analysis. ACM Sigplan Notices. ACM. (2006).

[23] Printezis, T., & Detlefs, D. A generational mostly-concurrent garbage collectorACM SIGPLAN Notices (2000, October 16). (Vol. 36, pp. 143-154). ACM.

[24] Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S., & Hundt, R. Google-wide profiling: A continuous profiling infrastructure for data centers. IEEE micro (2010), (4), 65-79.

[25] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G., et al. Profiling a warehouse-scale computer. Proceedings of the 42nd Annual International Symposium on Computer Architecture. ACM. (2015).

[26] Burton, F. W., & Sleep, M. R. Executing functional programs on a virtual tree of processors. Proceedings of the 1981 conference on Functional programming languages and computer architecture. ACM. (1981).

[27] Halstead Jr, R. H. Implementation of Multilisp: Lisp on a multiprocessor. Proceedings of the 1984 ACM Symposium on LISP and functional programming. ACM. (1984).

[28] Kumar, V., Frampton, D., Blackburn, S. M., Grove, D., & Tardieu, O. Work-stealing without the baggage. ACM SIGPLAN Notices. ACM. (2012).

[29] Xiaoning Ding, Kaibo Wang, Phillip B. Gibbons, and Xiaodong Zhang. BWS: balanced work stealing for time-sharing multicores. Proceedings of the 7th European Conference on Computer Systems (EuroSys 2012), pages 365-378, 2012.

[30] Barabash, Katherine et al. "A parallel, incremental, mostly concurrent garbage collector for servers." ACM Transactions on Programming Languages and Systems (TOPLAS) 27.6 (2005): 1097-1146.

[31] Lieberman, Henry, and Carl Hewitt. "A real-time garbage collector based on the lifetimes of objects." Communications of the ACM 26.6 (1983): 419-429.