Gang Scheduling Java Applications with Tessellation

Benjamin Le Jefferson Lai Wenxuan Cai John Kubiatowicz*

{benjaminhoanle, jefflai2, wenxuancai}@berkeley.edu, kubitron@cs.berkeley.edu
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720-1776

ABSTRACT

Java Virtual Machines (JVM) are responsible for performing a number of tasks in addition to program execution, including those related to garbage collection and adaptive optimization. Given the popularity of the Java platform and the rapid growth in use of cloud computing services, cluster machines are expected to host a number of concurrently executing JVMs. Despite the virtual machine abstraction, interference can arise between JVMs and degrade performance. One approach to prevent such multi-tenancy issues is the utilization of resource-isolated containers. This paper profiles the performance of Java applications executing within the resource-isolated cells of Adaptive Resource-Centric Computing and the Tessellation OS. Within this setup, we demonstrate how overall performance and JVM efficiency for the DaCapo benchmarks and Cassandra on the Yahoo Cloud Serving Benchmark degrade. We compare the effect of gang scheduling the threads of a cell together with using a simple global credit scheduler. We show that from the limited results we observed, there is no convincing evidence of gang scheduling yielding any kind of benefit, but also that a more complete evaluation may still be needed.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management – Scheduling; D.4.8 [**Operating Systems**]: Performance – Measurements, Monitors

General Terms

Multicore, parallel, quality of service, resource containers

Keywords

Adaptive resource management, performance isolation, quality of service

*The Parallel Computing Laboratory, UC Berkeley, CA, USA

1. INTRODUCTION

The Java Virtual Machine (JVM) abstraction provides a consistent, contained execution environment for Java applications and has played a significant role in Java's popularity and success in recent years. The JVM abstraction encapsulates many tasks and responsibilities that make development faster and easier. For example, by handling the translation of portable Java bytecode to machine code specific to the lower level kernel and system architecture, the JVM allows developers to write an application once and run it in a number of different environments. While there exist a number of JVM implementations, maintained both privately and publicly, in general, implementations adhere to a single JVM specification. We refer the reader to [16] for a complete specification, but summarize the roles of two key, highly researched components of the JVM: the garbage collector (GC) and the adaptive optimization component of the "just in time" (JIT) compiler [20].

Garbage collection is the process of cleaning up unused memory so that developers do not need to manage memory themselves. While this makes development easier and can drastically reduce the number of memory related bugs, garbage collectors are very complex and consumes resources, including time. Many techniques for garbage collection have been developed, from simple reference counting, to parallelized stop-and-copy, to generational garbage collection [17]. Each technique differs in how to locate "live" objects, when to run, whether and when execution needs to be halted, what memory needs to be touched, and how objects in memory should be moved.

The JIT compiler, on the other hand, is responsible for compiling segments of bytecode into native machine instructions. Modern JIT compilers are extended with adaptive optimization techniques [23], which attempt to optimize performance by dynamically compiling and caching these segments of code during run time. The result is a drastic performance improvement over simply interpreting the bytecode. Exactly how much compilation occurs before versus after a program begins executing and how much of a given program is interpreted varies across JVMs and significantly affects the performance characteristics of the program. While more advanced optimization results in higher performing code, as with GC techniques, it generally also requires more resources to perform, incurring higher overhead and impact on the program's performance.

Together, garbage collection and adaptive optimization require the JVM to perform many background tasks in addition to program execution. While this design works well

when there is a small number of Java applications on a machine, it may not scale well when there are many Java applications running concurrently on a single machine. This type of situation can often be found on the machines in cloud computing clusters. Given the ubiquity of Java based software applications, as demand for cloud computing and hosting services increase, understanding the types of multi-tenancy issues that may arise becomes critical to progress. For example, a machine in the cloud could be asked to host multiple instances of Hadoop File System (HDFS) [22], Hadoop MapReduce [4], and Spark [25], simultaneously, with each instance having a dedicated JVM. As all of these long-running JVMs must ultimately be multiplexed onto a single set of hardware, interference may arise between the tasks of each JVM. For certain applications, such as those with strict timing requirements or quality of service (QoS) guarantees, this interference may lead to unacceptable performance.

In addition to investigating the significance of the issues named above, we consider a potential solution made possible with the *Tessellation* operating system architecture and the *Adaptive Resource-Centric Computing (ARCC)* system design paradigm [6, 7, 18]. In ARCC, applications execute within stable, isolated resource containers called *cells*. In addition to implementing cells, the ARCC-based Tessellation kernel uses *two-level scheduling*, which decouples the allocation of resources to cells (the first level) from scheduling how these resources are used within cells (the second level). In this paper, we apply and evaluate gang scheduling [9] as a second level scheduling policy. In particular, we execute a single application in each cell and run multiple cells on a single, multi-core machine, gang scheduling each cell's resources together. By measuring the performance of these applications with and without gang scheduling, we show the effect of gang scheduling at mitigating interference for OpenJDK's HotSpot JVM.

The remainder of this paper is organized as follows. Section 2 provides background on the HotSpot JVM and the Tesselation architecture. Section 3 and section 4 describe our experimental setup and results with two different Java benchmark suites. Section 5 describes our results when using a gang scheduler. In section 6 we present a survey of related work. We discuss directions for future work in section 7 and conclude in section 8.

2. BACKGROUND

This section provides an overview of the relevant features of OpenJDK's HotSpot JVM and of the primary features of the Tessellation manycore operating system.

2.1 HotSpot

There currently exist two primary implementations of the HotSpot JVM. One is owned and maintained privately by Oracle and the other is an open source version maintained alongside the OpenJDK project. Despite being separately maintained, the two implementations are strikingly similar in methodology and performance. This is largely due to the fact that each stems from the same version made public by Sun Microsystems as well as the fact that the two sets of contributors overlap considerably. The primary differences between the two stem from Oracle's addition of several non-essential, auxiliary components. As such, and for lack of detailed documentation from OpenJDK, we present two key components of the Oracle HotSpot JVM here and make the, we believe valid, assumption that our discussion applies wholly to the OpenJDK implementation.

The HotSpot JVM is designed as a high-performance environment for Java applications with a focus on performance optimization. On top of implementing the features required by the aforementioned Java Virtual Machine specification, HotSpot provides a number of options and operating modes for tuning for different use cases. This focus is embodied in HotSpot's implementation of the garbage collector and adaptive optimizer. We now summarize the main features of these two components and describe how the synchronization characteristics of each make gang scheduling an appealing option.

2.1.1 HotSpot Garbage Collection

The garbage collector in HotSpot is a *generational garbage collector* [19] that partitions the heap into two generations: the *young generation* and the *old generation*. The idea behind this is that most allocated objects have short lives and do not survive more than a small number of garbage collections. Thus, efficiency is improved by primarily allocating from the smaller young generation and collecting this frequently while holding long-living, tenured objects in the larger old generation and collecting it less frequently. These significantly differing characteristics mean that the young and old generations can, and should, be collected using different collection algorithms in the interest of performance and efficiency.

The young generation is separated into an *eden space* and two *survivor spaces*. Most new objects are allocated from the eden space.[1] When that becomes full a *minor collection* occurs, which collects space from dead objects and moves them between the survivor spaces. At the end of the collection the only one survivor space contains active data. These are the live objects that survived collection but have not been promoted to the old generation.[2] HotSpot allows this collection to be performed serially or in parallel. While the serial version may be applicable for small scale, personal uses, we focus on the the parallel version, which is more appropriate and the default for the server class machines we are interested in. In both cases, minor collections are always *stop-the-world* collections that require application execution to be paused. As such, in the case of parallel collection, a natural barrier is exists at both the start and end of a minor collection, and a small amount of communication between threads is required to initially divide the work among threads.

In contrast to the young generation, the old generation is not divided; the entire space is used to hold objects that are large or long-living. When the old generation becomes full, a *major collection* occurs, during which both generations are collected, with the young generation collection typically occurring first. The collection of the old generation can be serial or in parallel and done in a stop-the-world fashion or concurrently during application execution. Again, while a serial *mark-sweep-compact* algorithm is available, we focus on two parallel algorithms which utilize multiple cores and reduce pause times.

The *parallel compacting (PC)* collector is a stop-the-world

[1]Some large objects that do not fit may be allocated directly to the old generation.

[2]If not enough space exists in the to space, the remaining objects are moved to the old generation.

collector that collects the old generation in three phases. In the *mark* phase, a set of "root" objects are divided among threads and live objects are marked, along with metadata for the data "region" containing them. The *summary* phase is performed serially and calculates the density of live data for each region so that the following parallel *compaction* phase can move and compact the data. Utilizing parallelism helps reduce pause times, especially when the size of the old generation is large. However, in some cases, even this reduced pause time is still too long. For such situations, HotSpot provides the *concurrent mark-sweep (CMS)* collector. Unlike the PC collector, the CMS collector only requires a small number of short pauses at certain synchronization points. CMS collection begins by a short pause called the *initial mark* phase that identifies an initial set of root objects. After this pause, the application resumes execution while a single GC thread identifies the live objects reachable by the root objects in the *concurrent mark* phase. Because the application is running during this phase, a second pause is made in the *remark phase* and multiple threads are used to find live objects that were missed. Afterwards, the application can resume while a single thread reclaims the garbage space in the *concurrent sweep* phase. The CMS collector helps avoid long pauses, but since it does not perform compaction, allocations and minor collections may take longer. This tradeoff can be favorable for applications with short pause time requirements. Both the PC collector and the CMS collector have synchronization points or barriers between each of their respective phases. In HotSpot, per-thread jobs are implemented so that they are largely independent and thus, only a minimal amount of synchronization is required within phases.

2.1.2 HotSpot Adaptive Optimization

Traditional JIT compilers perform compilation just before execution, and thus are limited in terms of compile time and the types of optimizations they make. For example, since Java supports features such as *dynamic class loading*, it cannot perform some types of aggressive optimizations such as extensive method-inlining. HotSpot's adaptive optimization component addresses several of these obstacles by initially interpreting the entire program and selectively compiling only certain parts of the code during run time [20]. This technique is based on the observation that for many programs, most of program's execution time is spent executing only a small portion of the program's code. Instead of spending valuable time and resources compiling code which is rarely executed, the compiler can detect "hot spots" in the code that are run very frequently and focus optimization on those parts. This includes, but is not limited to, compiling and caching these "hot spots." In addition, by *code profiling* or monitoring and gathering information about the program's execution patterns, HotSpot can perform some of the aggressive optimizations done by static compilers, including method-inlining, as well as new types of optimization techniques. One important example of this is *dynamic deoptimization*. Since execution patterns can change and new code can be loaded while the program is running, the HotSpot advanced optimizer can deoptimize (e.g. revert back to interpretation) and reoptimize the code in response.

Of course, performing this profiling, analysis, optimization consumes resources. Profiling itself incurs a relatively small amount of overhead. It consists largely of maintaining performance counters to monitor method and loop entries [13]. However, the analysis and JIT compilation portion of the optimizer can consume a significant amount of resources, depending on how the program is written and how stable the code profile is. This compilation and analysis is performed alongside the application. In addition, a fair amount of synchronization is required in order to dynamically modify code paths safely and efficiently.

HotSpot comes packaged with three choices for compilers: the client compiler, server compiler, and tiered compiler. The client compiler focuses on being fast and minimizing startup delays by performing mostly quick and simple local optimizations. In contrast, the server compiler is designed for server-like, long-running programs. It performs a number of complex, advanced optimizations that require extra time in order to gather the large amount of profiling data needed. The tiered compiler attempts to achieve "the best of both worlds" by initially using a client compiler to compile profiled code quickly. The client compiler ultimately feeds into a server compiler, which can use the compiled, profiled code to perform more advanced optimizations quickly. In this paper, we utilize the server compiler because we expect multi-tenant JVM issues to be more prevalent when the JVMs host long-running tasks. We choose the server compiler over a tiered compiler simply because it is currently much more widely used than the tiered compiler, which is relatively new.

2.2 Tessellation

Adaptive Resource Centric Computing (ARCC) is a design paradigm which establishes an interface between resources and applications through the abstraction of *cells* [7]. A cell is essentially a stable, consistent resource container for applications. A cell can also be used as a wrapper for a set of resources and exposes a service interface for that resource to other cells. Resources are allocated to cells and applications running within them are provided quality of service guarantees for those resources. Within ARCC, resource allocations to cells are automatically adjusted in a way to best meet application requirements.

The Tessellation OS architecture [6, 7, 18] implements cells using an approach called *space-time partitioning*, in which hardware resources are virtualized, "spacially" partitioned, and time multiplexed. Concretely, this means that resources such as CPU cores, memory, and service guarantees from other cells can be partitioned and shared by a number of processes. Tessellation's model for accomplishing this is through *two-level scheduling*, which separates resource allocation (first level) from how those resources are used (second level).

The cell abstraction offered by the ARCC and Tessellation align very well with the goals of the JVM abstraction. Perhaps most importantly, both models attempt to provide a stable and isolated execution environment for applications so they do not need to concern themselves with lower level system complexities. Moreover, the QoS guarantees, resource isolation and freedom of scheduling within cells in Tessellation are able to provide conditions near that of a dedicated machine. In the rest of this paper, we profile the effect of multiple, concurrent JVMs on the performance of applications with different types of workloads when running on an implementation of the Tessellation OS. We investigate the significance of inter-JVM interference and the cost/benefit of

preventing it by comparing the effectiveness of gang scheduling the resources within a cell (i.e. the resources provisioned to a given JVM) with using a simple global credit-scheduler [1].

3. DACAPO WITH XEN

The DaCapo benchmark suite represents a diverse set of open source, real world applications [5]. The applications in the suite are chosen to maximize coverage of the types of applications run in the wild and to be relatively easy to use and measure. These applications are also well known to put considerable pressure on the JVM garbage collector and to benefit significantly from JVM warm up. The benchmark harness supports multiple workload sizes, configurable number of iterations to run a benchmark, and the reporting of a performance-stable iteration.

We utilize the DaCapo 9.12 bach benchmark suite released in 2009, though we use only a subset of this suite for our experiments (avrora, jython, luindex, xalan). This subset was chosen based off which benchmarks ran without errors within our guest OS, OS^v [14]. In the interest of space, we showcase only the results for two of our benchmarks, avrora and xalan. The other two benchmarks exhibit results that are similar to and more pronounced in avrora and xalan. The characteristics of these 2 benchmarks are listed below.

Avrora: Avrora is a multithreaded application which simulates programs being run on a grid of AVR microcontrollers. Each microcontroller is simulated by a single thread. Avrora exhibits high fine-grained synchronization between simulator threads.

Xalan: Xalan transforms a XML document featuring the works of Shakespeare into HTML. Xalan is multithreaded, creating a thread for each available CPU. Threads process elements off a work queue in parallel.

3.1 Experimental Setup

Figure 1 showcases our experimental setup. Our setup uses the OpenJDK7 implementation of the Java platform. We deploy our JVMs in Virtual Machines running the OS^v 0.13 Operating System [14]. OS^v VMs are run as guest domains on the Xen 4.4 hypervisor [3]. The hypervisor runs on a 2-socket machine with 2 Intel Core i7-920 processors at 2.67GHz (4 cores per socket, Hyperthreading off, 8 total CPUs). Our test machine has 12GB of physical memory and runs Ubuntu 14.04.01. Each Xen guest domain runs with 6 vCPUs and is allocated 512MB of memory. We run Xen domains within a 6-CPU Pool while reserving 2 CPUs and 3GB of memory for Domain-0.

We run three experiments using DaCapo and Xen's default credit scheduler for CPU resources. First, we measure the overall performance of running JVMs in parallel. Second, we measure the pause times of a *stop-the-world* parallel garbage collector. Third, we explore the performance of HotSpot's adaptive optimizer during the warm up phase while running alongside multiple JVMs. For all experiments, the same benchmark is run across all concurrent JVMs using the default workload size.

3.1.1 Overall Performance

For each benchmark in our subset, we run 1, 2, 4, 8, and 16 JVMs in parallel, with maximum heap sizes of 64MB, 128MB, and 256MB. We warm up our JVMs by running the benchmark for some number of iterations. The number of warm up iterations is empirically determined by calculating the average number of iterations needed until performance stabilizes for that benchmark. We continue to run the benchmark until all JVMs complete at least 5 iterations following warm up. The execution times of these 5 iterations are recorded. We also record the percentage of CPU time each domain spends in the *concurrency hazard* run state using the xentrace and xenalyze tools. Concurrency hazard is the domain state where some vCPUs are running while the other vCPUs are waiting to be scheduled to run.

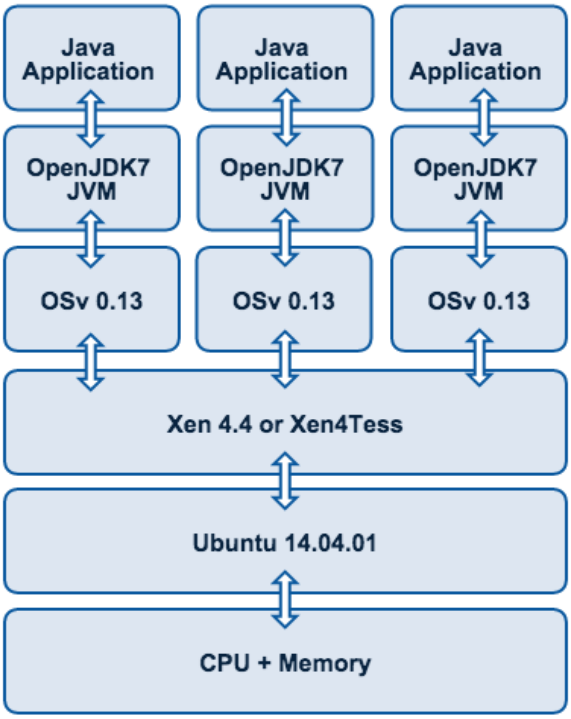

Figure 1: The application layering of our experimental setup.

3.1.2 Parallel Garbage Collector Performance

In the same setup as Experiment 1, we run the parallel JVMs with the *-XX:-UseParallelOldGC* flag turned on. This specifies that a parallel garbage collector be used for both the young and old generations. We record the GC pause times for both minor and major (full) collections that occur during the execution of the next 5 iterations after warm up.

3.1.3 Adaptive Optimizer Performance

First, for each benchmark, we run a single JVM in isolation with maximum heap sizes of 64MB, 128MB, and 256MB. Similarly to Experiment 1, we record the execution times of the 5 iterations following warm up. Next, we warm up a single JVM with the same heap sizes while the machine is under the load of 15 other JVMs running the same benchmark. After warm up, we kill the other JVMs and run the single JVM in isolation, again recording the execution times of the next 5 iterations. We repeat these steps for each combination of benchmark and heap size 5 times, recording

5 data points per combination.

3.2 Results

3.2.1 Overall Performance

We begin by profiling overall performance as we scale the number of concurrently running JVMs on our system. The left-most column in Figure 2 shows the overall total run time slowdown relative to 1 JVM. We calculate slowdown by dividing the mean total run time for the next 5 iterations after warm up for N JVMs over that for 1 JVM. For avrora and xalan respectively, performance is stable for up to 4 and 2 concurrently running JVMs. Because these benchmarks are quite old, *resource saturation* does not occur until multiple JVMs are deployed. After saturation, there is a roughly linear increase in slowdown.

In the center column, we graph the average fraction of CPU time spent in the concurrency hazard run state as we scale up the number of JVMs. We wish to explore whether the credit scheduler is naturally scheduling the vCPUs for our applications together, or in other words, imitating gang scheduling. We are most interested in the concurrency hazard run state because this is when vCPU clock skew occurs, something that can be reduced if vCPUs are gang scheduled together. We hypothesize that increasing the number of JVMs will increase the chance of being in the concurrency hazard run state. Our results show that this is true, the fraction of time spent in concurrency hazard spikes and then slowly dips off scaling to 16 JVMs. After investigating why this dip off occurs, we discovered that domains spend a majority of their time in the partial contention run state at 16 JVMS, up to 50% for avrora and 60% for xalan. Partial contention is when some vCPUs are waiting to be run while others are blocked, as opposed to running for concurrency hazard. We believe that the increase in time spent in partial contention is due to hitting IO bounds when scaling up the number of JVMs.

In the right most column, we graph the continous density function for iteration run times of all 5 iterations after warm up across all JVMs. We are interested in how variation for run time is affected as we scale up the number of JVMs. We only show CDFs for 128MB maxmium heap size because CDFs for 64MB and 256MB are very similar. For both avrora and xalan, we observe longer tails on both ends when running 16 JVMs in parallel. This indicates that the end user running these JVMs will experience more inconsistent performance under increasing multi-tenancy in the system.

Under gang scheduling, we do not expect overall performance to improve but we do believe that performance can be *more consistent* due to a more accurate abstraction of CPU resources.

3.2.2 Parallel Garbage Collector Performance

When a parallel garbage collector is utilized, synchronization is necessary at the start and end phases of garbage collection (GC) [20]. All application threads are paused and brought to a safe point before the GC event and all GC threads need to finish before the application threads continue running again. Scheduling interference may cause clock skew between these application and GC threads. We hypothesize that as we linearly increase the number of concurrently running JVMs, GC run times will increase non-linearly due to the overhead of blocking and waiting for these threads to synchronize.

Figure 3 shows the GC run time slowdown relative to 1 JVM. We calculate slowdown by dividing the total time spent in GC for the 5 iterations after warm up for N JVMs over that for 1 JVM. Like our slowdown graphs for overall performance, resource saturation occurs when 4 and 2 JVMs run in parallel for avrora and xalan respectively. After saturation, GC run time increases linearly, paralleling the results found for overall performance. The synchronization between threads during GC seems to not be fine grained enough for scheduling interference to be significant [9]. Work stealing for load balancing of GC also prevent GC threads from blocking and waiting for long times during collection itself [10]. Thus, we do not expect GC performance to improve with gang scheduling.

3.2.3 Adaptive Optimizer Performance

At run time, the HotSpot compiler performs extensive profiling of an application and uses that data to aggressively optimize machine code for the current execution environment [20]. We have demonstrated in section 3.2.1 that in the presence of multiple running JVMs, performance becomes more inconsistent for individual iterations. Thus, we hypothesize that if optimization occurs during a period of high load on the system, unpredictable performance may cause the adaptive optimizer to make sub-optimal, or even poor, decisions in the translation of bytecode to machine code for the system.

Figure 4 compares the performance of two different JVMs running a benchmark in isolation. Error bars with a height of one standard deviation are displayed on the graph. We calculate the mean total running time of the 5 iterations after warm up for 5 JVMs. For both avrora and xalan, there is no statistical difference between a JVM that has warmed up in isolation and a JVM that has warmed up with 15 other peers. The variation in iteration run times due to interference does not appear large enough to cause a JVM to optimize machine code differently. Any fine tuning of machine code is likely insignificant to performance compared to identifying and compiling hot spots in the first place. Thus, we do not expect the adaptive optimizer to benefit from gang scheduling.

4. CASSANDRA WITH XEN

Cassandra is a write-optimized distributed NoSQL DBMS implemented in Java originally designed by Facebook [15]. It provides linear scalability and high availability, and the proven fault-tolerance on commodity hardware makes it a strong candidate for cloud infrastructure. The experiment here attempts to investigate the interference and GC pressures that arise from running multiple Cassandra JVMs. This refers to real world scenarios, such as Amazon Dynamo [12], where multiple virtual nodes are running concurrently on a single physical node.

4.1 YCSB

To measure the performance of concurrent Cassandra instances, we use the Yahoo Cloud System Benchmark(YCSB) [8], to issue workload requests to Cassandra nodes. With YCSB, we are able to control the request type and workload distribution. We also manage the total record size and total number of requests to fit our machine capacity.

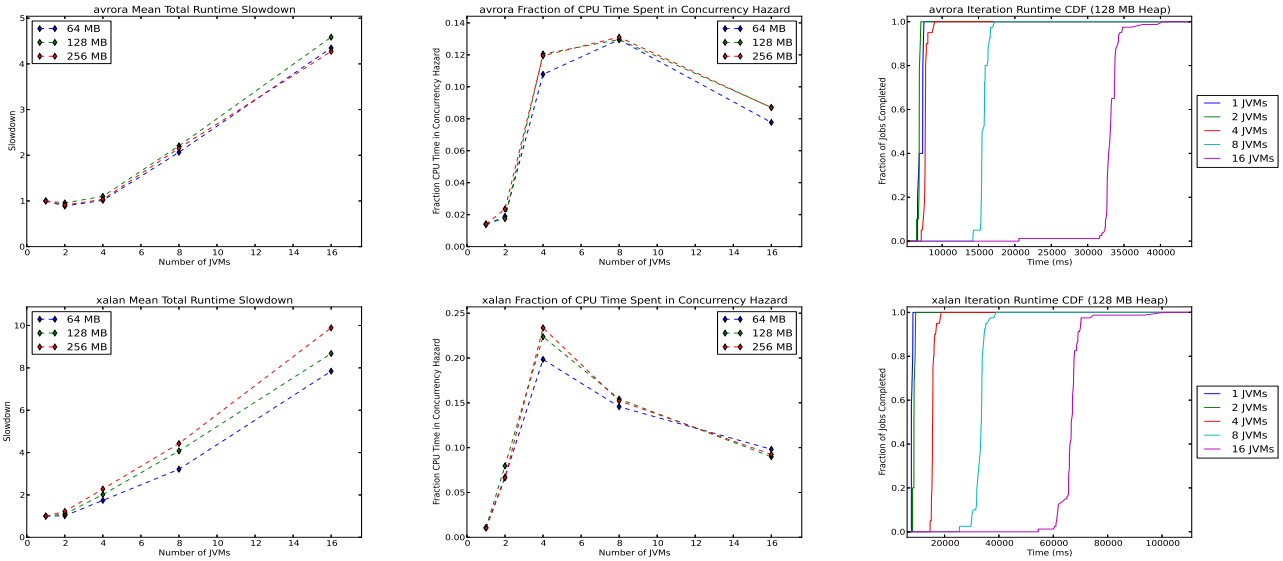

Figure 2: Run time slowdown relative to 1 JVM, CDF of iteration run times, and fraction of CPU time spent in *concurrency hazard* respectively.

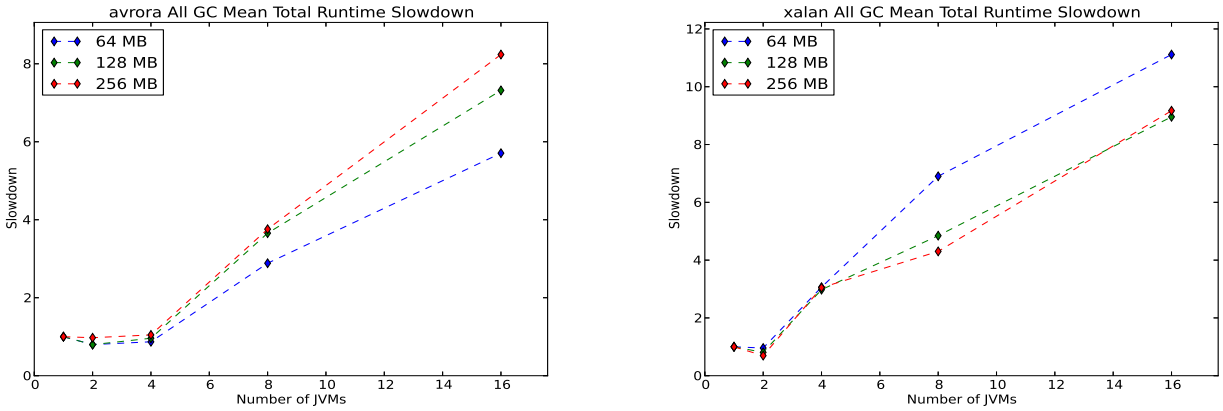

Figure 3: GC run time slowdown relative to 1 JVM.

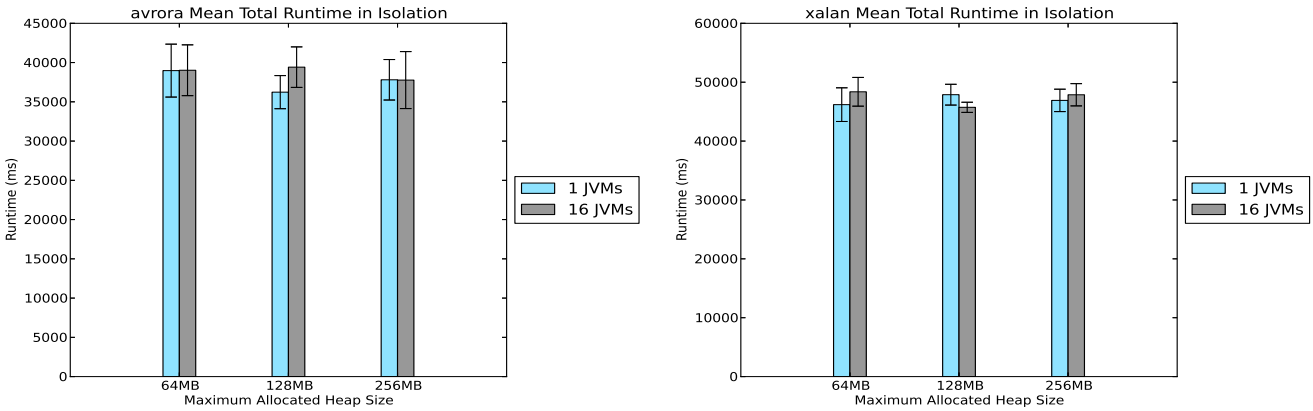

Figure 4: Mean total run time of JVMs warmed up in isolation and with 16 running JVMs in parallel.

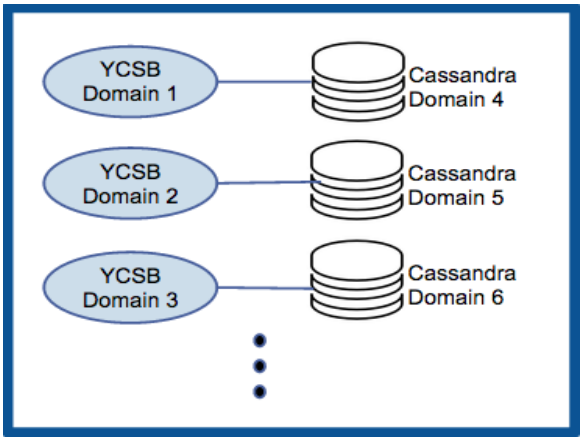

Figure 5: The set up of Cassandra and YCSB pairs

4.2 Experimental Setup

Our hardware setup for this experiment is identical to that in 3.1. We run both Cassandra and YCSB as OS^v guest domains upon Xen hypervisor on the same machine and infrastructure. Figure 5 shows the architecture of our experiment. We pair each YCSB domain with one Cassandra domain to ensure that each Cassandra instance will receive the same number of requests and thereby maintain consistent workload across multiple Cassandra instances and eliminate load balancing as a potential source of variance. All domains are configured in a private LAN through a Xen bridge, to minimize the network influence on request latency and throughput. Given the high memory demand from Cassandra DBMS and the 12GB memory capacity of the test machine, we limit the memory allocation for each Cassandra and each YCSB domain to 2GB and 512MB respectively. This means we can scale up to at most 4 Cassandra plus 4 YCSB domains. All domains run with 6 virtual CPUs on the default Xen credit scheduler. Due to the above memory constraint, we cap the record count in each Cassandra database to 5000, with 10 fields each. The first field is the primary key, and the other 9 fields are of type `varchar`. We first measure the overall performance by benchmarking multiple Cassandra instances concurrently with two different types of workloads. We then measure the impact of increasing JVM count on the *Concurrent-Mark-Sweep* garbage collector.

4.2.1 Overall Performance

Our experiments evaluate 1, 2, and 4 instances of Cassandra-YCSB "pairs," which corresponds to 2, 4, and 8 JVMs respectively. We set the request count to 100,000 and run two kinds of workloads. The first one is update heavy, consisting of 50% read and 50% update. This type of workload is typical for applications such as a session store. The second workload involves a large portion of read-modify-write operations. Related applications include a user database, where user records are read and modified by the user or to record user activity. This workload is 50% read and 50% read-modify-write. Both workloads use zipfian distribution when picking the record to update. This long-tail distribution, where the frequency of an item is inversely proportional to its rank, reflects the real world fact that most operations only act upon a small set of records. [2]. Each field is up to 100 bytes in length for a total data size of 1KB. For each benchmark, we first run a load phase to populate the database, then run one iteration of run phase to warm up our JVMs. After that, we run 5 iterations of run phase and record the overall and operation-level latency/throughput. A final value is calculated as the average of those measurements across our multiple JVMs.

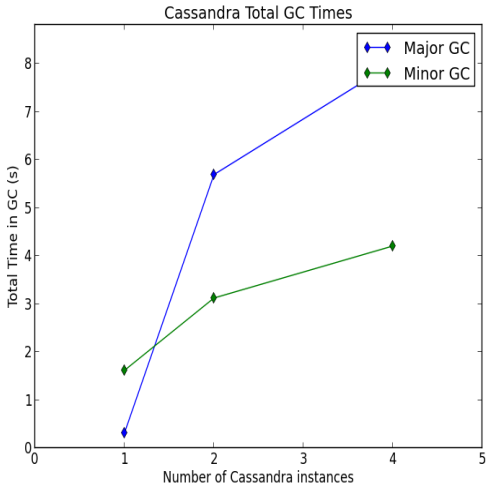

Figure 7: GC total time increases under same workload with additional Cassandra instances

4.2.2 Concurrent-Mark-Sweep Performance

Using the same setup as the above experiment, we run our Cassandra JVMs with the *-XX:+UseParNewGC* flag on to enable the CMS collector. We record the GC time of both minor and major (mark and sweep) collection phases during the execution of the 5 iterations of run phase after warm up.

4.3 Results

After running both workloads, we observed similar performance degradation relative to the number of Cassandra instances. Due to space limitations, we show the results of the read-modify-write workload here and hold them as representative of both.

4.3.1 Overall Performance

We showcase the individual 99^{th} percentile latency plots for read, update, and read-modify-write operations in Figure 6. The plots show a nonlinear increase from 1 to 2 to 4 "pairs" of Cassandra-YCSB instances. When compared with writes, reads show a slightly larger degradation from 2 to 4 pairs than from 1 to 2 pairs. We expect that the robustness advantages of writes relative to reads is due to the write optimization of Cassandra, which utilizes *memtable* and *sstable* [15]. We hypothesize that the slowdown from 2 Cassandra-YCSB domains to 4 stems from the fact that in the 2 pair case, there are 4 JVMs in total, and all domains can be scheduled concurrently on 6 physical CPUs. However, with 4 domains and 8 total JVMs, not all domains can run at the same time. There is no guarantee that the scheduler would schedule the Cassandra instance and YCSB instance in each domain together. Thus, it is possible to have one side blocked while the other side is running. We discuss this issue further in section 5.

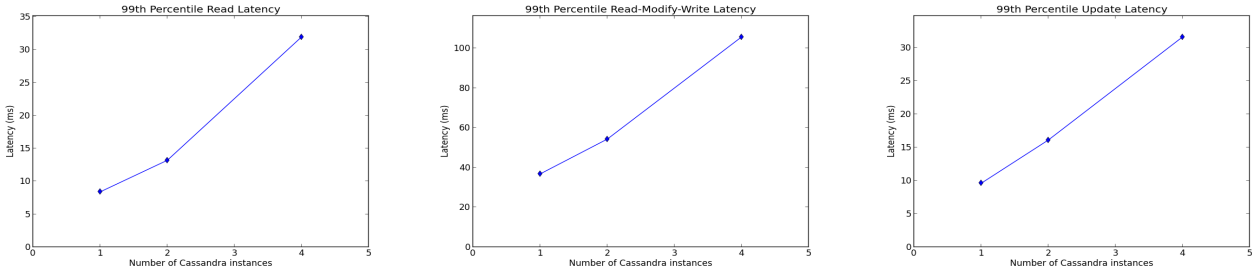

Figure 6: Operation-level 99 percentile latency increases as adding extra Cassandra instances for read, read-modify-write, and update.

4.3.2 Concurrent-Mark-Sweep Performance

Figure 7 shows the increase of both major and minor GC time due to the additional Cassandra instances running. As the CMS collector uses the same parallel collector for the young generation from our DaCapo experiments, it is not surprising to see a pattern of minor GC slowdown identical to the one in section 3.2.1. We find the run time of minor GC increases linearly as a function of the number of concurrently running JVMs. This similarity confirms the early idea that synchronization between threads during garbage collection does not seem to be fine grained enough for scheduling interference to show a significant impact in our experiment.

5. GANG SCHEDULING

A custom Xen hypervisor (Xen4Tess) featuring a prototype implementation of a gang scheduled CPU Pool was used for our gang scheduling experiments. The gang scheduler runs vCPUs of Xen domains one after the other with a fixed 50ms scheduling quantum. Unfortunately, we were unable to install a stable version of Xen4Tess on our test machine. Furthermore, limitations in domains shutting down or doing anything other than running at full-bore prevented us from running our full experimental suite for both credit and gang scheduling in time. Before running an experiment with a new combination of benchmark and heap size, old running VMs are shutdown, causing the whole machine to hang.

Nevertheless, we were able to get a similar setup to Figure 1 up and running on one of our workstations to run small DaCapo experiments. In this setup a Xen4Tess hypervisor runs on a single-socket machine with an Intel Core i5-3570k processor at 3.4GHz (4 cores per socket, Hyper-threading off, 4 total CPUs) and 16GB of physical memory. Each Xen guest domain runs with 3 vCPUs and is allocated 512MB of memory. We run Xen domains within a 3-CPU Gang Scheduled Pool while reserving 1 CPU and 7GB of memory for Domain-0.

5.1 Dacapo

Using gang scheduling and this new machine, we re-performed the experiment in section 3.1.1 to measure overall performance for each of our benchmarks for only the 16 parallel JVMs, 128MB maximum heap size combination. However, we were unable to use xentrace and xenalyze to observe run states for gang scheduled domains as unexpected run state transitions would occur due to the gang scheduler.

Figure 8 shows the average overall total run time of avrora and xalan for both credit and gang scheduling. We graph the iteration run time standard deviations as well. In section 3.1.1, we hypothesize that overall performance will not improve much with gang scheduling. However, we did not expect that the gang scheduler would be so detrimental to performance. Avrora total run time increased 1.9x over credit scheduling while xalan increased 1.2x. When profiling the domain run states in section 3.1.1, we did discover that vCPUs block more often when scaling up to 16 JVMs. Because gang scheduling is unable to reassign pCPUs to different domains when the current domain's threads block, many CPU cycles are wasted that could otherwise be occupied with work from other domains. The over-provisioning of CPU resources from gang scheduling also affects performance negatively because resource saturation does not occur until multiple JVMs are run in parallel.

We are surprised that run times for individual iterations become more unpredictable under the affects of gang scheduling, disproving our initial guess. We are unsure why this behavior is happening but perhaps there is large variation in how long vCPUs block for within a scheduling quantum. Blocking may occur early in the quantum, resulting in a lot of wasted CPU cycles, or not at all. Overall, gang scheduling does not appear to provide any benefit that we could observe over the default credit scheduler for our DaCapo benchmarks.

5.2 Cassandra

Unfortunately, we were unable to repeat any Cassandra experiments with gang scheduling. However, gang scheduling appears promising in improving YCSB performance [9]. In particular, gang scheduling can schedule client and server processes together. Take the previous Cassandra and YCSB experiment for example. If the gang scheduler were aware of their need to communicate with each other, performance can benefit from scheduling the pair together. This avoids the situation where one side of communication is blocked while the other side is active.

6. RELATED WORK

Our experiences with gang scheduling reflect some of the findings in [21]. In it, the authors describe how the benefits to gain from gang scheduling are not worth the complexity in implementing it for current workloads. The work asserts that a set of conditions, including latency sensitivity, fine-grained synchronization, burstiness, multiple high-priority workloads, must be met for gang scheduling to yield any significant benefits. However, it is possible that an increas-

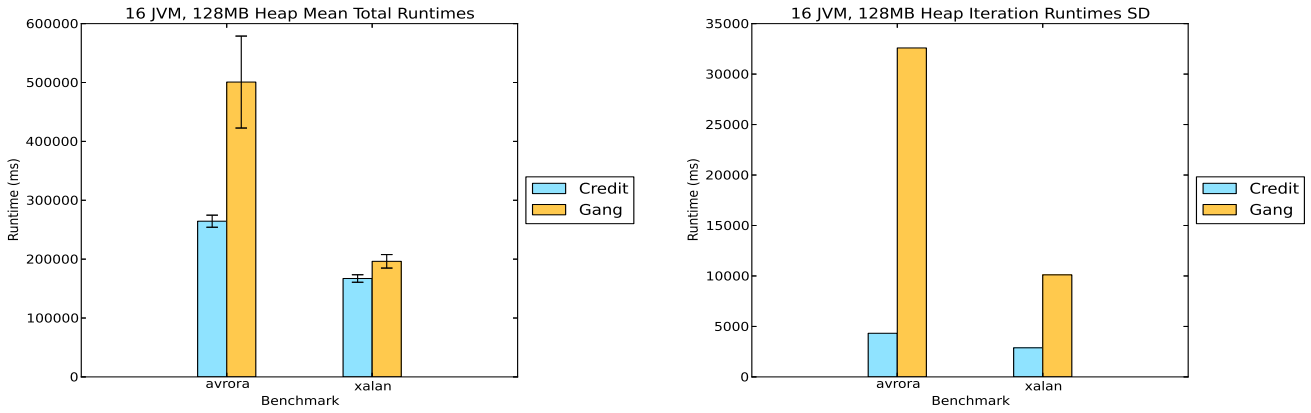

Figure 8: Overall performance and iteration standard deviation comparison of credit and gang scheduling

ing number of future application workloads may fall into this category. We complement this work by evaluating how well the the combined workloads of Java applications and their associated JVMs fit and compare to such workloads. We describe our plans for evaluation of different workloads in section 7.

Callisto [11] is a resource management layer for parallel runtime systems. It utilizes dynamic spatial partitioning and co-scheduling in order to reduce the interference. The findings and accomplishments in Callisto is very similar to what we hope to achieve in this work. However, while Callisto targets general parallel runtime systems with CPU-intensive workloads, we focus specifically on JVM-specific tasks for applications with a wide variety of workloads.

Rather than strictly gang scheduling vCPUs together, the *ESX* hypervisor by VMWare uses a relaxed co-scheduling algorithm that ensures that the skew on each vCPU does not grow too large in comparison to its peers [24]. vCPUs that are lagging behind can be run individually to catch up while those that advanced too far stop and wait. Although more overhead is needed to keep track of vCPU progress, the scheduler benefits by having more flexibility in scheduling choices.

7. FUTURE WORK

As we are currently limited by the instability of Xen4Tess with gang scheduling on our test machine, we plan to replicate each of our DaCapo experiments with gang scheduling as soon as such a more stable implementation is available. This will enable us to more accurately compare benchmark behavior under gang scheduling with our base case.

We also plan to scale these experiments up on a machine with more resources and execute more JVM instances (32, 64, 100) to extract more evidence to verify our measurements. Similarly, we plan to also examine more types of workloads, including synthetic ones. We expect to find different results for workloads with finer grained synchronization and/or strict timing requirements. We wish to also run experiments using different combinations of garbage collection algorithms for young and old generations.

To verify our hypothesis in Section 5.2, we plan to explicitly gang schedule domains which need mutual communication. We also intend to explore the Tessellation event-triggered cell [7], an ideal model for hosting services. It supplies flexible event-handling as well as good responsiveness and resource utilization. Both fit well with the role of Cassandra in our experiment.

8. CONCLUSION

By utilizing a modified version of the Xen hypervisor that implements the Tessellation architecture, we provided baseline measurements on the performance of the DaCapo benchmarks and Cassandra's performance on the YCSB benchmark. We observed that for DaCapo, overall slowdown and GC overhead scaled linearly with increasing number of JVMs, but predictability in performance suffers. Multitenancy was also not detrimental to HotSpot's adaptive optimization. With Cassandra, we observed that 99% latency measurements grow non-linearly with increasing number of JVMs. Unfortunately, due to resource constraints, we were only able to extract a limited set of data points for this baseline. Thus, while we observed the trends above, we can not say with reasonable confidence that they hold when scaling beyond 4 JVMs. Our measurements on a early-implementation of gang scheduled cells indicate that for the DaCapo benchmarks, performance drops significantly using the gang scheduler. Consistency in benchmark iteration run times surprisingly suffer as a result of gang scheduling as well. There are a number of possible explanations for this, including variance in time spent in blocked state and bugs in the implementation. Future work in exploring different Java applications and JVM components is needed to explore when gang scheduling may be beneficial for JVMs.

9. ACKNOWLEDGMENTS

We would like to thank Martin Maas for advising us throughout our project along with Nathan Pemberton for writing the gang scheduler implementation for Xen, and Eric Roman and Kostadin Ilov for providing and setting up test machines.

10. REFERENCES

[1] E. Ackaouy. The Xen credit CPU scheduler. In *Proceedings of*, 2006.

[2] L. A. Adamic and B. A. Huberman. Zipf's law and the internet. *Glottometrics*, 3(1):143–150, 2002.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[4] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene. apache. org/hadoop*, 11, 2005.

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

[6] J. A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz. Resource management in the Tessellation manycore OS. *HotPar10, Berkeley, CA*, 2010.

[7] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, et al. Tessellation: refactoring the OS around explicit resource containers with continuous adaptation. In *Proceedings of the 50th Annual Design Automation Conference*, page 76. ACM, 2013.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[9] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[10] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 229–240, New York, NY, USA, 2013. ACM.

[11] T. Harris, M. Maas, and V. J. Marathe. Callisto: co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems*, page 24. ACM, 2014.

[12] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *In Proc. SOSP*. Citeseer, 2007.

[13] P. Hohensee. The HotSpot Java Virtual machine. `https://www.cs.princeton.edu/picasso/mats/HotspotOverview.pdf`.

[14] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association.

[15] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[16] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[17] R. D. Lins. Garbage collection: algorithms for automatic dynamic memory management. 1996.

[18] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 10–10. USENIX Association, 2009.

[19] S. Microsystems. Memory management in the Java HotSpot virtual machine, 2006. `http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`.

[20] Oracle. The Java HotSpot performance engine architecture, 2013. `http://www.oracle.com/technetwork/java/whitepaper-135217.html`.

[21] S. Peter, A. Baumann, Z. Anderson, and T. Roscoe. Gang scheduling isn't worth it ... yet. Technical Report 745, Department of Computer Science, ETH Zurich, November 2011.

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[23] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *ACM SIGPLAN Notices*, volume 36, pages 180–195. ACM, 2001.

[24] VMWare. VMware®vSphere™: The CPU scheduler in VMware ESX®4.1, 2010. `http://www.vmware.com/files/pdf/techpaper/VMW_vSphere41_cpu_schedule_ESX.pdf`.

[25] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.