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 applications run within Java Virtual Machines (JVM). As an application runs, the JVM performs many parallel background tasks for general housekeeping reasons including garbage collection and code profiling for adaptive optimization. While this design works well to provide isolation when there is a single or small number of Java applications running on a single machine, in practice it is common to find a large number of Java applications running concurrently on a single machine. For example, a machine could be running multiple instances of HDFS, Hadoop, and Spark, simultaneously, with each instance having an associated JVM. As all of these JVMs must ultimately be multiplexed onto a single set of hardware, interference among the large set of parallel tasks arise. There is little published literature documenting the causes of this interference or how to deal with it. In this paper, we determine the specific sources of these interferences and show how running these applications on top of a Tessellation-integrated Xen hypervisor addresses these issues and reduces interference. We evaluate the performance of these âĂIJTessellatedâĂİ machines in comparison with machines running bare Linux when running a large number of JVMs.

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 is a significant factor in why Java has enjoyed so much popularity and success in recent years. 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, they adhere to a single JVM specification. We refer the reader to [6] for a complete specification, but summarize the roles of two key, highly researched components of the JVM: the "just in time" (JIT) compiler and the garbage collector (GC).

The JIT compiler is responsible for compiling segments of bytecode into native machine instructions. Modern JIT compilers perform adaptive optimization [9], which optimizes performance by dynamically compiling and recompiling 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 optimizations result in higher performing code, they generally also require more resources, including time, to perform, causing the program's performance to suffer for a period.

Garbage collection is the process of cleaning up unused memory so that developers do not, and generally can not, manage memory themselves. While this makes development easier and can drastically reduce the number of memory related bugs, garbage collectors are very complex and, as with the JIT compiler, consumes resources. Many techniques for garbage collection have been developed from simple reference counting, to parallelized stop-and-copy, to generational garbage collection [7]. 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 may need to be moved.

Together, garbage collection and adaptive optimization require the JVM to perform many tasks background tasks in addition to program execution. While this design works well when there is a single or small number of Java applications running on a single machine, in practice it may be desirable to have a large number of Java applications running concurrently on a single machine. For example, a machine could be running multiple instances of Hadoop File System (HDFS), Hadoop, and Spark, simultaneously, with each instance having an associated JVM. As all of these JVMs must ultimately be multiplexed onto a single set of hardware, significant interference may arise between the tasks of each JVM. In particular, when applications require timing or quality of service (QoS) guarantees, this interference may lead to unacceptable performance.

One potential solution to this problem is made possible with the Tessellation operating system architecture and the Adaptive Resource-Centric Computing (ARCC) system design paradigm [2, 3, 8]. 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 [4] as a second level scheduling policy. In particular, we execute a single application and JVM 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. Sections 3 and 4 describes our experimental setup and results with two Java different benchmark suites. In Section 5 we present a survey or related work. We discuss directions for future work in Section 6 and conclude in Section 7.

2. BACKGROUND

2.1 HotSpot

2.2 Tesselation

3. DACAPO

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 [5]. OS^v VMs are run as guest domains on the Xen 4.4 hypervisor [1]. A custom Xen hypervisor (Xen4Tess) featuring a prototype implementation of a gang-scheduled CPU Pool is also used for our gang-scheduling experiments. 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 serviced by 6-CPU pool and is allocated 512MB of memory. We reserve 2 CPUs and 3GB of memory for Domain-0.

We only run a subset of the Dacapo benchmark suite in our experiments (avrora, jython, luindex, lusearch, xalan). This subset was chosen because other benchmarks within the suite have bugs when running within OS^v.

We run two separate experiments using Dacapo. First, we measure the overall performance difference between running

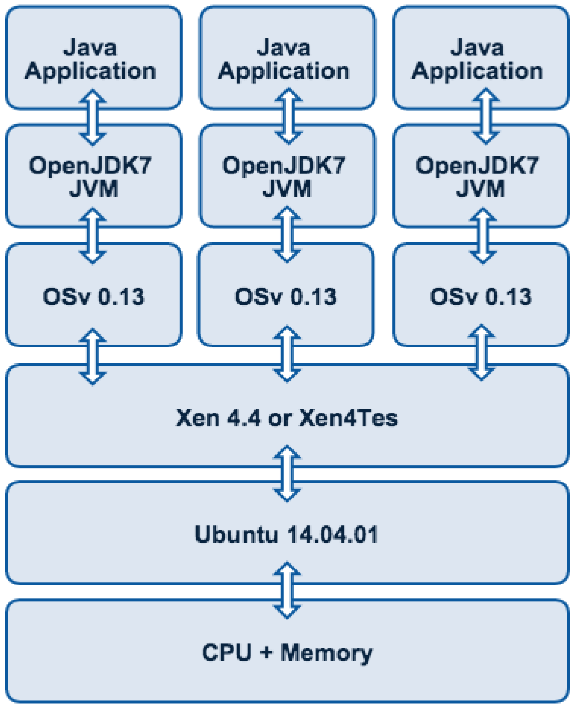

Figure 1: The application layering of our experimental setup.

JVM's in parallel with and without gang scheduling of CPU resources. In our second experiment, we wish to explore the performance of the JIT compiler's code profiler and optimizer in the warmup phase during parallel JVM load.

3.1.1 Experiment 1: Overall Performance

For each benchmark in our subset, we run 1, 2, 4, 8, and 16 JVMs in parallel with maximum heapsizes 1x, 2x, 4x, and 8x times the minimum heapsize for the benchmark. The minimum heapsize for each benchmark is empirically determined by reducing the heapsize until an out of memory error occurs. We warmup our JVMs by running the benchmark for some number of iterations. The number of warmup iterations is empirically determined by calculating the average number of iterations needed until performance stabilizes. The execution times of the next 5 iterations after warmup are recorded and an arithmetic mean is reported as the benchmark runtime for each JVM.

3.1.2 Experiment 2: JIT Compiler Performance

First, we run a single JVM in isolation with maximum heapsizes 1x, 2x, 4x, and 8x times the minimum heapsize for the benchmark. Similarly to Experiment 1, we record the execution times of the next 5 iterations after warmup and compute an arithmetic mean. Next we warm up a single JVM with the same heapsizes under the load of 15 other JVMs running in parallel. After warmup, we kill the other JVMs and run the single JVM in isolation; again computing the arithmetic mean of the execution times of the next 5 iterations. We repeat these steps for each combination of benchmark and heapsize 5 times recording 5 data points per combination.

Table 1: Standard Deviations of Dacapo Benchmarks

Benchmark	Standard Deviation	Variance
avrora	1000	100

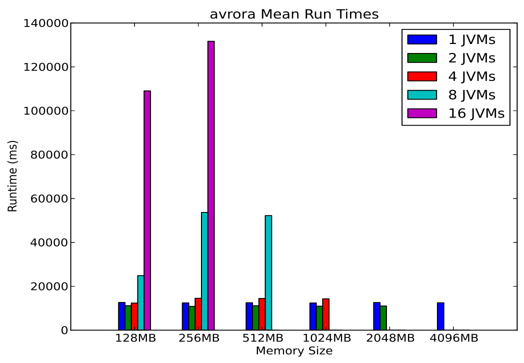

Figure 2: A sample graphic (.eps format) that has been resized with the `epsfig` command.

3.2 Results

4. YCSB

4.1 Experimental Setup

4.2 Results

5. FUTURE WORK

6. RELATED WORK

7. CONCLUSION

8. ACKNOWLEDGMENTS

Martin Nathan Eric

9. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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, 2014. USENIX Association.

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

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

[8] 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.

[9] 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.

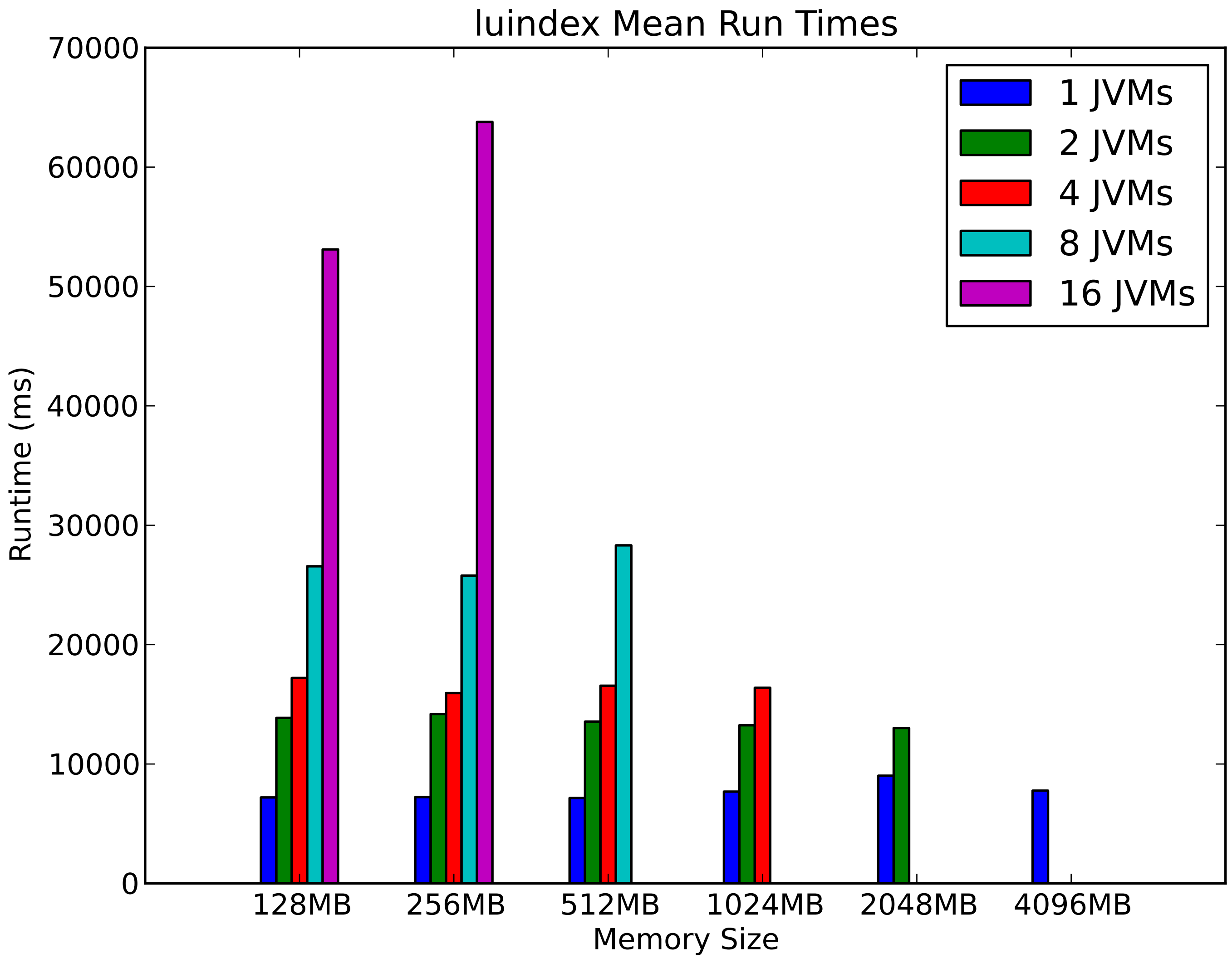

Figure 3: A sample black and white graphic (.eps format) that needs to span two columns of text.