2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems

Evaluation of Multi-Core Scalability Bottlenecks in Enterprise Java Workloads

Xavier Guerin, Wei Tan, Yanbin Liu, Seetharami Seelam and Parijat Dube
IBM T. J. Watson Research Center
Yorktown Heights, NY, USA

***Abstract*—The increasing number of cores integrated into modern processors is blurring the line between supercomputers and enterprise-grade servers. Therefore, the same attention to lock contention bottlenecks must be given to Java-based business workloads as it is given to massively parallel, high-performance computing applications, especially when it comes to characterizing global trends that would ease the transition of today's code base to tomorrow's parallel configurations. This paper first presents the characteristics of a typical Java-based business application software stack and examines the locking contentions that can appear at each level of that stack. Second, it presents scalability evaluation of three enterprise-grade, Java-based workloads and details the lock contention founds. Third, it summarizes the results of our findings, emphasizing the need for a streamlined methodology for lock-contention analysis of enterprise Java workloads.**

I. Introduction

Multi-core designs are progressively replacing high-frequency operating architectures, hence bringing new software challenges. In particular, Java-based enterprise software is not able to efficiently exploit this newly available parallelism due to several years of development and multiple layers of abstraction. Although future applications can possibly be designed and developed to efficiently exploit multi-core features using new programming models like X10 [44], it is challenging and sometimes infeasible to re-design existing commercial applications from scratch.

A more realistic approach is evaluate scalability bottlenecks in each layer of the application's stack and incrementally implement solutions. This is a tedious process, not only because it requires to deal with a stack depth of at least six layers but also because Java-based applications rely on two distinct execution domains, the Java domain supported by the Java virtual machine and the system domain supported by the operating system, unlike applications written using non-interpreted languages such as C/C++. This particularity makes the tracing and analyzing of bottlenecks complicated, especially when the frontier between these two domains is blurred by *just-in-time* compilation techniques. Therefore, the discovery of common bottlenecks would be an asset for future analysis.

This paper evaluates the scalability of three arbitrary Java-based enterprise workloads – IBM WebSphere ILOG Business Rules Management System (BRMS), PDF generation, and IBM Cognos Chart Generation Service (CGS) – on a multi-core system. The goal of this evaluation is to characterize the scalability bottlenecks of these applications, provide solutions, and identify commonalities. The rest of the paper is organized as follows. Section II describes the protocol of our evaluation. Sections III, IV, and V respectively analyze the IBM Websphere ILOG BRMS [18] workload, the PDF generation workload, and the Cognos CGS workload. Section VI discusses issues at the Java virtual machine level which hinder scalability due to interactions of the application with the operating system stack. In Section VII we compare performance of multi-thread execution with multi-JVM for CGS workload to support the value of lock-contention analysis for enterprise Java workloads. An overview of existing works related to Java workload scalability is presented in Section VIII. Section IX concludes this paper.

II. Evaluation Protocol

This section describes the hardware and software environments as well as the methodology and the profiling tools that compose the protocol of our analysis. We believe this protocol to be straightforward and generic enough to be applied to any Java application on any platform to characterize performance and analyze scalability.

A. Hardware Environment

We used an IBM POWER7-based blade server for our experimentations. The system consists of two POWER7 processors and 256 GB of system memory. The POWER7 processor is the latest instance of the POWER architecture [32]. Each POWER7 processor consists of 8-cores for a total of 16 POWER7 cores for the blade. Each core operates at a 3GHz clock frequency and features a 32 KB level-1 cache for data and instructions respectively and a 256 KB level-2 cache. A 32 MB level-3 cache is shared among the 8 cores of a processor.The POWER7 architecture also supports various hardware simultaneous multi-threading modes. In the scope of this evaluation, we disabled this support to avoid jittering related to thread scheduling [11] and focus on the raw scalability of the workloads on the physical cores.

B. Software Environment

We used SUSE Linux Enterprise Server 11 as operating system loaded with the version 2.6.32.19-0.3 of the Linux [42] kernel on our blade. We used the latest stable Java6-compatile release of IBM's J9 Java Virtual Machine (JVM) for our experimentation. For each benchmark, we used a single JVM

1526-7539/12 $26.00 © 2012 IEEE
DOI 10.1109/MASCOTS.2012.43

instance using the GenCon [20] policy and sufficient amounts of memory heap in order to avoid any garbage collection activity (a minimum heap size of 24 GB and a minimum nursery size of 16 GB).

C. Methodology

Initial measurements were collected using the default parameters for each workload. Then, we conducted our analysis using a top-down methodology based on good-faith: each layer of rank n (topmost) of the software stack was profiled with the hypothesis that each other layer $n-1 \ldots 1$ is scalable and free of lock contention until the last layer has been reached. This approach is based on the fact that a scalability hindrance present in a software layer of rank n is more likely to hide, either by construction of by effect, hindrances present in layers of lower ranks $n-1$. As such, locating secondary hindrances can prove difficult and the resolution unfruitful *a priori*.

For each benchmark, we choose to execute the application on the maximum available number of cores and gradually increase the number of application threads from 1 to 16, and collect the results for each step. This choice guarantees that each thread related to the workload's execution can run freely on its own processor core. We also present the theoretical results one would have gotten were the application performance scaling perfectly with the number of cores. Perfect scalability entails linear throughput increase and constant latency up to hardware limits. Perfect scalability is reached today by using multiple JVMs, usually one per core (see e.g., [39]). But the waste of system resources attached to this method makes a good case to analyze the scalability of a workload using a single JVM [2].

When the collected results show signs of bottlenecks, we first evaluate application-related issues using IBM's WAIT tool [1], [19]. Other tools such as JProfiler [13] could also be used. Secondly, we evaluate the JVM itself using IBM's TProf [22] and JProf [21] tools, both compatible with Oracle HotSpot as well.

III. ILOG Business Rules Management System

Information technology has become a mandatory tool for companies to manage their business. As such, IT departments are facing increasing rates of software modification to keep their supported applications up to date with constantly changing business policies. Such modifications are usually risky especially on large and complex applications.

A Business Rules Management System (BRMS) proposes to separate the business policies of the enterprise applications into business rules. Externalizing business policies as business rules facilitates the design, deployment, execution, and management of business logic without putting the original system in jeopardy.

A. Business Rules

A business rule expresses a business policy in a form both understandable to business users and executable by a rule engine. Thus, a business rule is both a statement that describes a certain aspect of a business policy and an executable entity that can be called from an application. Here is an example of the kind of business policy that can be expressed as a rule:

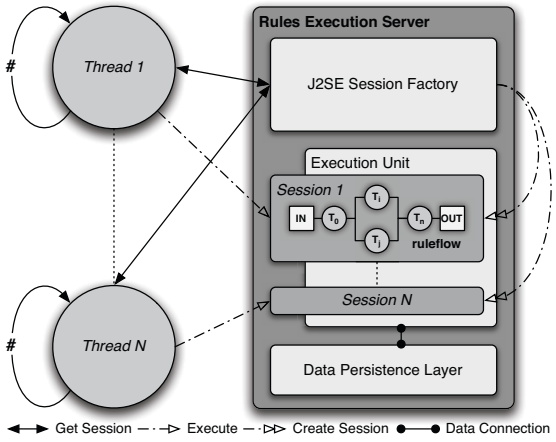


Fig. 1. ILOG Benchmark Execution

A person whose age is over 50 gets 35% off his subscription.

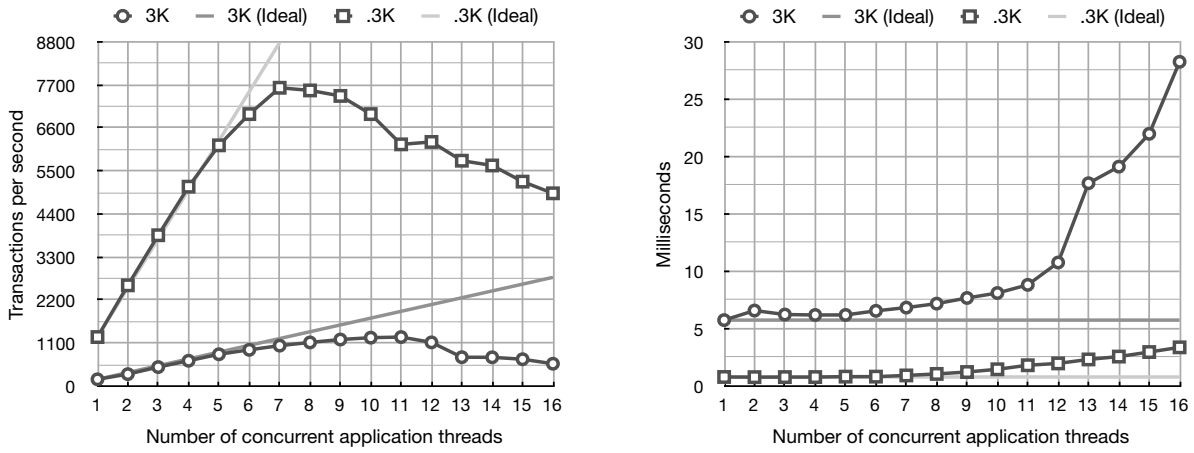
The definition of a rule is a two-step process. First, a vocabulary required to express the policy as a conceptual object model is formalized. Then, the logic of the business policy is represented using if-then statements. For instance, the above business policy can be implemented with the following business rule:

If
`the person's age is over 50`
Then
`remove 35% off his subscription price`

In this form, the business logic can be packaged as a rule set and called from the application code as a single entity. Therefore, changes to the business policy do not require changes to the application or process code. In the rest of the paper, the **If** part of a rule is referred to as its *condition statement* and the **Then** part of a rule is referred to as its *execution body*. The format used to store a persistent version of a rule is called Business Action Language (BAL). The compilation of the BAL rules generates technical versions of these rules. Called Intermediate Rule Language (IRL) and defined as a Java dialect, this technical version is used by the Rule Execution Server (RES) to instantiate rule execution engines.

B. Rules Execution Server

The RES provides various methods to executing business rules. It encapsulates a more primitive entity, called the Rule Execution Engine (REE), in charge of the raw execution. When executed, its first job is to load and compile the rule set, and bootstrap an execution engine on this rule set. The number of concurrent executions can be increased to improve both the throughput and the average latency of the application. The working data of the rules can be fetched either from a file or a database or accessed directly from memory. The RES executes



(a) Throughput

(b) Latency

Fig. 2. ILOG BRMS Scalability Results

a rule set according to a control and data flow graph named *ruleflow*. A *ruleflow* can accept and produce data (*in* and *out* nodes in Figure 1). Such data is called *ruleset parameters*. Each node of a *ruleflow* is called *task* and corresponds to one instance of a rule execution engine. A *ruleflow* can be defined with mutually exclusive tasks to reduce the number of rules evaluated during the execution.

A rule execution engine can rely on three different algorithms to execute the rules: *sequential*, *fastpath*, and *Rete*. The *sequential* algorithm is a greedy algorithm that, for each rule in the rule set, sequentially executes its condition statement and its execution body if the condition was successful. The *fastpath* algorithm is an extension of the *sequential* algorithm that factorizes execution bodies of rules sharing identical condition statements. The *Rete* algorithm is an efficient pattern matching algorithm specifically designed for rule execution systems and first published in a working paper by Charles Forgy in 1974 and later formally in [14]. It is preferably used instead of *sequential* or *fastpath* when the execution of the rule set modifies the working memory (e.g. when the evaluation of a rule depends on the result of the execution of another rule).

The benchmark relies only on the *sequential* algorithm for the following reasons: 1) *sequential* and *fastpath* are the most used algorithms and 2) the only difference between *sequential* and *fastpath* resides in the way the BAL representations of rules are generated and not in the way rules are executed. Consequently, the execution patterns of both *sequential* and *fastpath* are equivalent.

C. Scalability

The benchmark used to gather the results presented in this paper is a simple market segmentation application that count the number of people that fits in a given category. The RES was used as a standalone Java application and not as part of the Websphere Application Server (WAS) to focus only on the rules execution logic and to avoid any interference. The application uses two rule sets, one of 300 rules and one of 3000 rules, and executes them on a memory-based, self-generated database of customers.

The customer objects are created directly and are not fetched using Structured Query Language (SQL) queries. Each customer has the following attributes: *gender* (male or female), *interests* (clothing, cosmetics, ...), and *product style* (classic, country, or trendy). The matching operation of each rule contains an arbitrary combination of these attributes. The execution operation of each rule is rather straightforward in that it simply increments a counter.

The considered application takes as arguments the number of execution threads to run in parallel as well as the number of execution iterations for each thread. The execution of the benchmark, depicted in Figure 1, takes place as follows. Initially, a benchmark loop with the given number of iterations (# in Figure 1) is instantiated for each requested thread. Then, the benchmark loop creates a connection with the RES, populates its local, in-memory database, and sends an execution request to the RES with the in-memory database as rule set parameter. Finally, the application cleans up the threads after all iterations are complete. The execution threads gather the number of Transactions Per Second (TPS) as well as the average rule execution latency for its run.

The TPS and latency results of the benchmark are presented in Figure 2 (with respectively .3K for the 300-rule rule set and 3K for the 3000-rule rule set). As shown in Figure 2(a), the throughput increases in a linear fashion for both rule sets to increase in a sub-linear fashion after five concurrent threads and then decreases after seven threads for the 300-rule rule set and eleven threads for the 3000-rules rule set. The latency results (shown in Figure 2(b)) are consistent with the throughput results. Up to five concurrent threads, the average rule execution latency is close to ideal for both rule sets to rapidly increase afterward. Each curve in Figure 2(a) has two interesting moments: the moment where the curve starts to increase sub-linearly and the moment where the curve starts to drop. For the 300-rule rule set curve, the first moment is

at 5 threads and the second moment is at 7 threads. For the 3000-rule rule set, the first moment is also at 5 threads and the second moment is at 11 threads. Our investigation shows that performance impacts for both rule sets stem from identical issues.

The first performance drop can be explained by a contention inside the rule execution server, in the connection manager itself. Each time the creation of a session is requested by the application, the connection manager takes a lock global to its own structure until the session is successfully created. As such, it is not too surprising that this lock would be contended since the creation of sessions is located on the critical path of the benchmark's execution. The second performance drop can be partly explained by the combination of the contention in the connection manager and a new contention found inside the implementation of the `setWeekCountData` method of the `Calendar` class, from the `java.util` package. A fix for that issue was provided and will be included in the next J9 release.

IV. PDF Document Generation

The second benchmark used in this study is an emerging workload. In an effort of limiting both the extraneous cost and the ecological impact of printing out report documents, the usage of Portable Document Format (PDF) [25] documents to replace said paper-based reports becomes more and more widespread. As the generation of such documents is often automated in batch for consolidation and archiving, generators need to be highly scalable to respond to any level of demand. Our discussions with many customer-facing colleagues reveal that typically customers leverage freely available packages such as iText [26], Apache FOP [4] and PDFBox [5], GNU JPDF [15], jFreeReport [31], or BFO PDF [9] to enhance their applications with report generation capabilities. Besides, virtually all reports nowadays include different kinds of charts for displaying tabular data. JFreeChart [30] or the BFO Java Graph libraries [8] are common examples of libraries providing such functionality.

A. Benchmark

Among these solutions, iText and jFreeChart are apparently the most used packages for dynamic PDF generation and manipulation. Therefore, we decided to base our benchmarking application on them. iText is a Java library used for creation and manipulation of PDF documents from Java applications. The iText library provides the functions to generate and manipulate PDF documents such as adding bookmarks, page numbers and watermarks, splitting, merging and manipulating pages, and adding digital signatures. iText is open source and is available both for Java and C#. We used jFreeChart to enhance PDF reports with graphical charts, including bar and pie charts that can be either 2-D or 3-D. This library provides sets of APIs to create professional quality charts for many output types including PDF. The PDF generation logic used in this benchmark has been extracted from the PDF generation extension of the DayTrader [3] benchmark presented in [37] and organized in such a way that its execution should theoretically be scalable: each PDF generation thread has its own execution stack and no data is shared at the application level. Therefore, a very special emphasis is made on the lower layers of the software stack than the application layer, namely the PDF generation and chart creation middleware, the JVM, and the operating system.

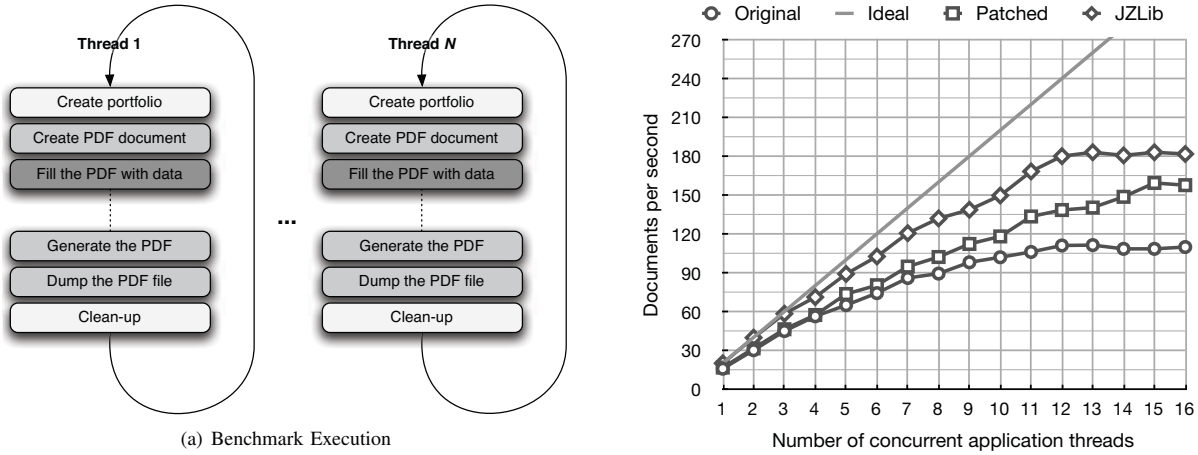
B. Scalability

The PDF generation application takes as arguments the number of execution threads to run in parallel as well as the time of execution in seconds. The execution of the benchmarks, depicted in Figure 3(a), takes place as follows. Initially, a benchmark loop constrained with a timer set to the given number of seconds to run is instantiated for each requested thread. Then, as shown in Figure 3(a), the benchmark loop: creates a portfolio using the DayTrader logic, creates a PDF document using the iText logic, fills the PDF with text and charts, generates the PDF document, dump the PDF file, and clean-up the created objects. Each execution thread keeps track of the number of PDF documents it generated and the benchmark uses these numbers to produce, at the end of its run, a throughput number in terms of generated PDF document per second. The throughput results of the benchmark are presented in Figure 3(b). In this figure, three values are presented: an *original* value, a *patched DuctusRenderer* value, and an *Ideal* value. The *original* value corresponds to an execution of the benchmark using the original version on the various middleware. The *patched DuctusRenderer* value corresponds to an execution of the benchmark using a patched version of the `DuctusRenderer` class from the `sun.java2d` package where a class-wide contention was removed. This patch is the result of previous investigations and it will be included in the next Java7-compatible J9 release. The results presented in Figure 3(b) show that the throughput increases in a linear fashion up to three concurrent threads then increases in a sub-linear fashion and finally flattens.

C. Lock Contention Issues

We discovered two outstanding lock contentions in two different Java system packages: `java.util.zip` and `sun.dc.pr`. The first lock contention is located in the native implementation of the `deflateByte` method of the `Deflate` class. By default, the iText PDF generator compresses each element added to a PDF document using the standard Java deflater mechanism. In addition, the benchmark uses a high level of compression to ensure realistic execution conditions, hence putting the deflater on the critical path of the execution.

It is not too surprising to observe some contention on a facility present on the critical path of a multi-threaded execution. What is surprising, however, is that each generation thread uses a different PDF generator, which in turn uses a different deflater object. As such, the contention in the native implementation of the `deflateByte` method cannot be intuitively explained. To circumvent that issue, we replaced



(a) Benchmark Execution

(b) Scalability Results

Fig. 3. PDF Benchmark and Scalability Results

the deflation occurrences in the iText implementation using the default deflation facility by their counterpart proposed in JZlib [29], an open-source implementation of the zlib library in pure Java. This modification has three benefits: 1) JZlib is functionally equivalent to the Zip implementation present in the Java class library, also based on zlib, 2) having a full access to the source code allowed us to potentially improve on it if needed, and 3) a pure Java implementation of a piece of software present on the critical path of the execution saves on Java Native Interface (JNI) calls overhead [40]. We repeated the execution of our benchmark using that modification. The results collected are presented in Figure 3(b) under the name *JZLib*. The first observation one can make is that the application modified to use JZlib performs better than the one relying on the stock implementation of Zip. The second observation is that although *JZlib* results are closer to the *Ideal* curve they are still slowly attenuated as the number of concurrent threads increases, to reach a ceiling after 12 threads. The data collected showed that switching to the JZlib eliminated the lock hidden in the deflater implementation, explaining the increase in performance.

Subsequently, the contention on the remaining lock in the `sun.dc.pr` package increased, explaining the final shape of the *JZlib* curve in Figure 3(b). This lock can be found in the native implementation of the `PathFiller` method, used in various graphic operations of the `sun.java2d` package. Calls to that package ultimately originate from the jFreeChart library. Here again, the contention cannot intuitively be explained as each thread, working on a separate PDF document, is working with its own graphic context and therefore should not share any graphic-related structure with the other threads. In an effort of thoroughness, we replaced the jFreeChart library with another Java-based graph library, the BFO Graph library [8], in the hope to see a different kind of behavior. Our measurements showed identical results, the BFO library being also crippled with the lock in the native implementation of the `PathFiller` method.

V. Cognos Chart Generation Service

IBM Cognos Business Intelligence (BI) [17] is a software suite for enterprise-scale reporting, analyzing, scorecarding and monitoring. As a typical J2EE application with a multi-tired architecture, Cognos BI builds on a service-oriented architecture (SOA) [45] which uses platform independent communication protocols such as XML (Extensible Markup Language), SOAP (Simple Object Access Protocol), and WSDL (Web Services Description Language).

Nowadays, business customers, from executives to managers, have an emergent need to deal with a wide range of reporting requirement – from professional reports with a large audience to ad-hoc ones with a small number of users. For example, a report can be a summary of last year's gross margin, or a prediction of next year's value. Chart generation is an important feature in Cognos BI, producing charts and figures to be used in various reports including PDF, Microsoft Excel and HTML. The Chart Generation Service (CGS) is the Cognos BI component that takes a chart specification as input, renders the result and returns the result as an image.

A. Benchmark

The execution flow of CGS is depicted in Figure 4. When the CGS service starts, it creates a socket listening to a port and a J2SE Session Factory managing a session pool. Each client thread sends its request to the CGS server via this socket, and the session factory either creates a new session or picks up an idle one from the pool to process the request. The request processing, i.e., the generation of a chart, consists of three steps. Initially, the chart specification is parsed from a XML format to an internal one used by the execution unit; then the chart is rendered as a binary byte stream; in the third step this byte stream is transformed into an ASCII string using Base64 encoding and inserted into a SOAP envelope as the response.

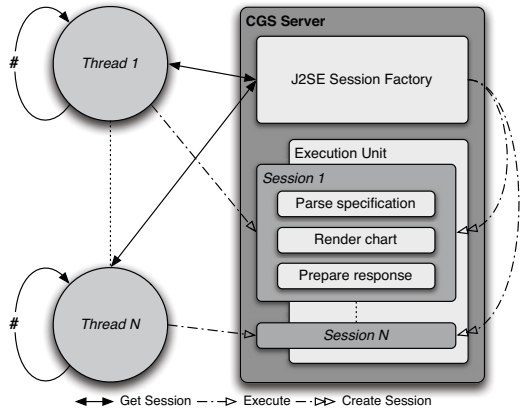


Fig. 4. CGS Benchmark Execution

The benchmark used to gather the results presented in this section is a report that aggregates the gross margin of a fictional company, categorised by product line and geographic region, and displays the result in two pie charts respectively. CGS is used as a standalone Java application and not as part of any application server to focus only on the chart generation logic. To avoid the interference with database, before we start this benchmark we query the Cognos database and make the chart specification ready to be sent to CGS – i.e., there is no database access during the running of this benchmark. The chart specification defines the chart type (for example, column, bar, pie or a mixture of them), layout, font and color scheme, and the actual data to fill in. The client to drive CGS defines the number of threads to initiate and the number of execution iterations for each thread. CGS creates a session pool with the same number of sessions as parallel clients connecting to it.

The execution of the benchmark, depicted in Figure 4, takes place as follows. Initially, a benchmark loop with the given number of iterations is instantiated for each client thread; then, each thread creates a connection to CGS and sends an execution request with the chart specification; finally, the application cleans up the threads after all iterations complete.

B. Scalability

The execution threads gather the number of Transactions Per Second (TPS) as well as the average latency for all runs, which are presented in Figure 5(a). In this figure, three values are presented: an *original* value, a *Text patch* value, and the *Ideal value*. The original value corresponds to an execution of the benchmark using the original version of CGS. The text-patch value corresponds to an execution using a patched version of CGS by removing a JVM-wide class lock (to be discussed later). The ideal value corresponds to the performance if CGS scales perfectly from one to sixteen threads. As shown in Figure 5(a), in the original case, the throughput increases in a sub-linear fashion before six threads, changes slightly from six to eight threads and then decreases after that. The latency results (shown in Figure 5(b)) are consistent with the throughput ones. Up to six concurrent threads, the average chart generation latency grows slowly and rapidly increases after eight.

C. Lock Contention Issues

We identified that a contention in class `javax.swing.TimerQueue` is the cause of the contention. CGS used class `JTextComponent` in package `javax.swing.text` to display texts in a chart. `JTextComponent`, like many other Swing components, has an associated `Timer` object managing delayed or recurring tasks (for example, to blink a displayed text, or update a progress bar regarding the percentage toward completion). `TimerQueue` is a singleton class to manage a queue of timers by the order in which they will expire. WAIT traces illustrate that, each time when we access a `Timer` from the singleton `TimerQueue`, it has to call its static `sharedInstance()` method to retrieve itself. Therefore when a lot of `Timers` request the `TimerQueue`, it becomes a bottleneck.

A fix of CGS was provided by replacing the `JTextComponent` class with a non-Swing `java.awt.font.TextLayout` class, and changing the detailed implementation accordingly. This patch will be included in the next CGS release. Data from Figure 5 shows that, while the original implementation scales in a sub-linear fashion and reach its peak throughput at eight threads, the patched, lock-free implementation scales to sixteen threads in an almost-linear fashion. Concurrently, we conducted the same evaluation on an eight CPU Intel Xeon 3.00GHz server using Oracle JRE 1.6 and **observed the same behavior**.

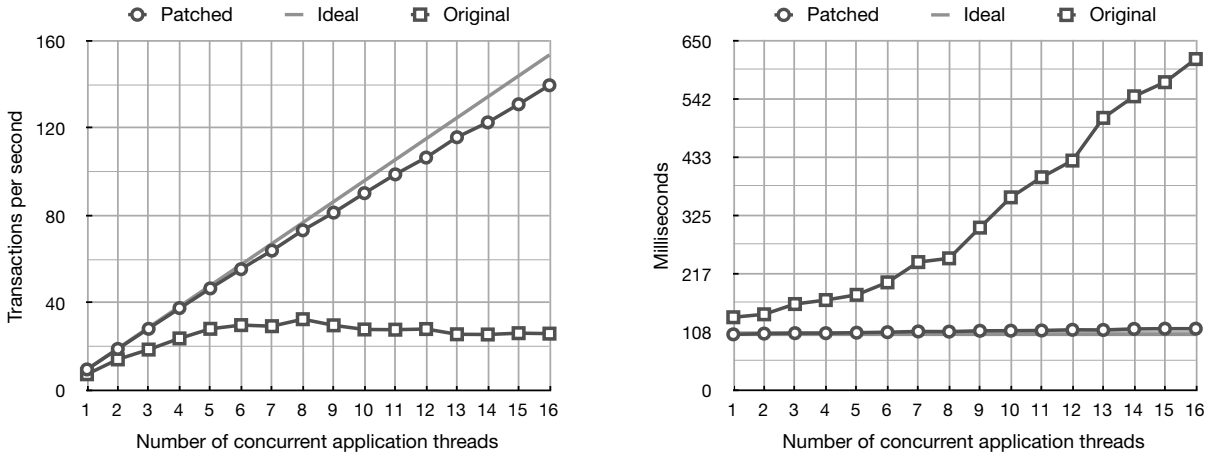
VI. Sotfware Adherence Issues

In this section, we detail the issues hindering scalability inside IBM's J9 Java Virtual Machine. The first section, specific to the ILOG BRMS application, describes how the application behavior impacts J9's JIT compilation engine. The next two sections, common to Java applications, deals with J9's three-tier locking mechanism and the impact of J9's system threads on scalability.

A. Impact of the Application on J9's JIT Compilation Engine

We used TProf in coordination with JProf and its Java Lock Monitor tool to profile the activity of the JVM itself, and found a contention point in a monitor inside the JIT compiler. The contention on that monitor is particularly noteworthy and stems from the way ILOG's RES produces executable rules from a rule set and the J9's multi-threaded JIT compilation engine. Both techniques will be covered in a later paragraph.

1) Generation of executable rules from a rule set: In a rule set, rules are stored in a non-executable form (see Section III-A). For the execution unit of the RES to execute them, the RES translates them on-demand into Java *bytecode* using technologies such as the Apache Byte Code Engineering Library [6] or the ASM library [36]. While the translation from BAL to Java *bytecode* is mostly done in a proactive fashion when the RES loads the rule set, this operation often overlaps the execution of the rules for large rule sets.



(a) Throughput

(b) Latency

Fig. 5. CGS Benchmark Scalability Results

2) J9 JIT engine multi-threaded architecture: In recent versions of the J9 JVM, the JIT engine has been split into multiple threads, one *Sampler* thread and multiple *Compiler* threads, to take advantage of massively multi-core processor such as POWER7. The *Sampler* thread gathers profile information for executed Java methods and orders them from "very cold" to "very hot" into a list. Concurrently, *Compiler* threads probe this list in search for potential just-in-time compilation candidates, usually "warm" to "very hot" functions - their "temperature" indicating the level of optimization to apply. Similarly, any "temperature" upgrade triggers a recompilation of the upgraded method.

The primary consequence of executing large rule sets is the generation of as many executable methods as rules available in the rule set, which will become, *in fine*, as many JIT compilation candidates since each method will eventually be executed multiple times. Since generated methods are shared amongst the various execution threads of the application, the raising in "temperature" of a given methods is even more rapid as the concurrency level of the application is high.

A higher rate of "very hot" methods leading to a higher pressure on the JIT engine, contention starts to appear on the JIT's shared structures, especially the list of JIT compilation candidates for which the contention was found, increasing the latency of the JIT engine which, in turns, increases the average latency of the application itself.

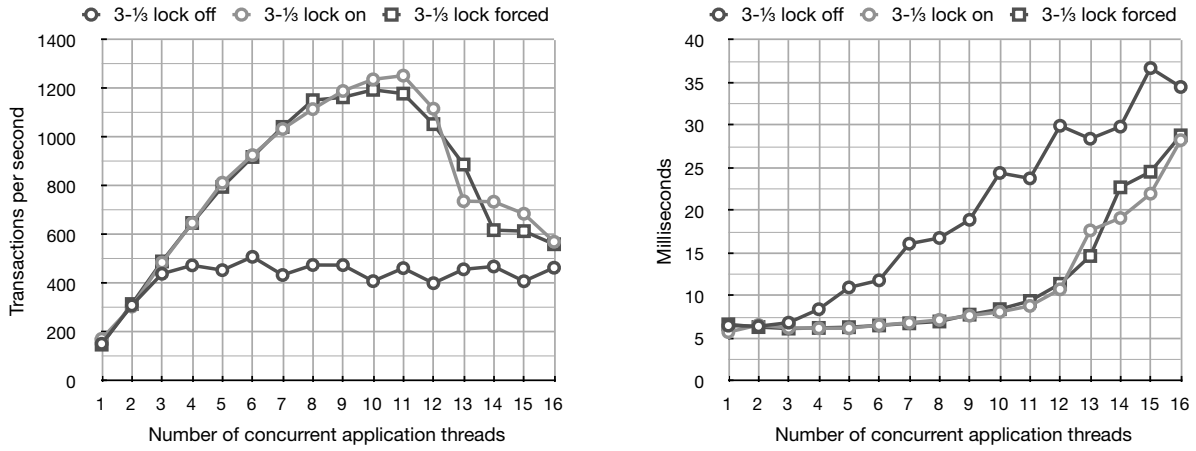
B. The Three-tier Locking Strategy

When dealing with locks, both costs of going to the operating system kernel to suspend the thread and then later waking it up when the lock is available can be significant. Even if the lock is available, going to the OS kernel to check its status can represent a significant overhead.

If the lock is not currently available but will be soon, it is often more efficient for the thread to simply "spin" for a short time until the lock is available. Three-tier locking is used in the J9 JVM to optimize object monitors, raw system monitors, and garbage collection spinlocks. Three-tier locking [12] uses techniques that allow to efficiently check locks when there is low contention. This denomination describes a spin strategy with three tiers of incrementally larger back-off between attempts to get the lock. This strategy has usually been beneficial until the introduction of the Completely Fair Scheduler (CFS) [34] as the sole and only scheduler of the Linux kernel since the 2.6.23 release. Indeed, internal benchmarks conducted on applications using WAS showed diminished performance with the CFS and the three-tier locking strategy when the number of application threads greatly outnumbers the number of available hardware processing units (processor cores, or hardware threads when SMT is available).

The identified cause of such performance diminution is the way CFS implements the `sched_yield` operation and how this operation is heavily used in the three-tier locking strategy. Disabling three-tier locking (each $*Iterations$ value set to 0) or forcing the CFS to use a retro-compatible implementation of the *scheduler yield* operation (through the `sched_compat_yield` configuration option in the kernel section of `procfs`) restores the initial level of performance.

However, this behavior has not been observed on well balanced workloads (where the number of application threads is less than or equal to the number of processing cores) for which the lack of three-tier locking proves to be critical. Results demonstrating this fact for the ILOG BRMS benchmark with the 3000-rules rule set are shown in Figure 6(a) and 6(b). In these figures, three values are presented: a *3-tier locking off* value, a *3-tier locking on* value, and a *3-tier locking forced* value. The *3-tier locking off* value represents the results of an execution using the CFS scheduler without the *scheduler yield* compatibility mode. The *3-tier locking on* value represents the results of an execution using the CFS scheduler with the *scheduler yield* compatibility mode. The *3-tier locking forced* value represents the results of an execution using the CFS



(a) Throughput

(b) Latency

Fig. 6. 3-Tier Locking Impact on ILOG BRMS Scalability

scheduler without the *scheduler yield* compatibility mode but with three-tier locking parameters equivalent to those set by the JVM when it is used without the CFS or when it is used with the CFS and its compatibility mode set.

Figure 6(a) gathers the throughput result for each execution case while Figure 6(b) gathers the latency results. While the difference between the results obtained with three-tier locking and those obtained without three-tier locking makes the benefits of such a strategy obvious, the results obtained with the *scheduler yield* compatibility mode activated and those obtained with forced locking parameters are very similar. This similarity shows that well-balanced workloads can benefit from both three-tier locking and the new CFS yield strategy without performance impact.

C. Impact of the JVM Threads

Last but not least, the very existence of threads dedicated to system tasks inside the virtual machine has an impact on the scalability of a Java-based application, especially when the activity of these system tasks is connected to the behavior of that application. The two subsystems of the J9 JVM making use of concurrent threads are the JIT compilation engine and the garbage collector.

As seen in section VI-A2, the JIT compilation engine uses one sampling thread and one to four compilation threads for a total of up to five concurrent threads. The activity of these threads is tightly coupled with the variation of the set of methods to execute and the variation of the application's execution pattern.

On one hand, if the set of methods of a given application and its execution pattern are roughly the same during the lifetime of that application (which is the general case), then the activity of the JIT compilation engine will have a minimal impact on the application over its lifetime, with peaks of activity located at the beginning of the application's execution. On the other hand, if the set of methods of a given application and its execution pattern greatly vary during the lifetime of that application (as it is the case for the ILOG BRMS application), then the activity of the JIT compilation engine will have a dear impact on the application, especially on its scalability as four to five processing units will be mostly occupied by the JIT compilation engine.

We tuned the ILOG BRMS benchmark differently to highlight the impact of the JIT activity on volatile execution patterns. For this execution, we used a bigger set of rules (12K rules) to increase the volatility of the execution path. We also constrained the number of concurrent application threads to eight. The performance increase for big rule sets being linear for up to eight concurrent threads, we have the guarantee that no lock contention is taking place.

We also constrained the number of available processing units to eight which corresponds to the maximum number of application threads, and twelve which corresponds to maximum number of application threads plus four processing units for the JIT sampler and compilers. The results show that the throughput starts dropping as soon as application threads and JIT compilation threads overlap, ending with 23% drop in throughput at eight concurrent threads.

Similarly, garbage collection can have an impact on an application's scalability. As we purposely configured the JVM used in our benchmarks to avoid any interference due to GC activity, our analysis does not cover such an impact. However, studies in [41] and [20], [12] are good starting points for the reader interested in these considerations.

VII. Discussion

The value of our investigation on identifying and removing lock contention in enterprise Java applications can be questioned by using an alternative multi-JVM strategy. That is, to bring up multiple Java virtual machines each of which run an instance of the same application. By this means the multiple working threads are separated into multiple JVMs (i.e., processes) so that lock contention diminishes. Despite of its simplify and inter-process isolation, this strategy has several

disadvantages. First of all, an additional external load balancer is needed to redirect incoming requests to multiple processes. Secondly, using multiple JVMs imposes larger memory footprint. Using multiple JVMs implies the duplication of method area (which stores class metadata such as the runtime constant pool, field and method data, and the byte code for methods), and JIT code and data cache.

We run the same Cognos CGS applications in a multi-JVM mode. That is, instead of running multiple treads on a single JVM, we run multiple JVMs and put one thread in each of it. The load balancer is in another machine sending workloads to each JVM. Even the overhead of load balancer is not counted on: our experiment shows that multi-thread mode is 11% better compared to multi JVM one.

In terms of memory footprint, CGS consumes 560 MB more memory in multi-JVM mode, when 16 threads are tested. 40% of this 560MB memory space is made up of duplicated method area and 60% the JIT cache data. The saving of memory by running a lock-free CGS in multi-thread mode is modest (560MB), because CGS is a lightweight J2SE program. An earlier work shows that, by removing lock contention and consolidating multiple instances of heavy-weight J2EE application server into one, 30 GB of memory was saved [2]. Hence, it is clear that efforts to removing lock contentions still make lot of sense: this enables workload consolidation into a single JVM, obtains better application throughput per byte memory, and increases resource utilization for useful work.

VIII. Related Work

Workload scalability is affected by different factors like hardware configuration, software design and configuration, network latency, database access. There are a number of studies on the performance analysis and scalability of Java applications on different systems (see, e.g., [10], [27], [43], [23], [24], [35]).

There have been many works looking at micro architectural features and their effect on Java workload scalability. They focused on identifying the effect of hardware configurations on workload scalability like number of cores, sockets, threads per core, L1 and L2 cache sizes and multi-processor architectures (SMP, CMP). The work presented in [10] analyzed JVM scalability on symmetrical multiprocessing (SMP) systems with number of processors and application threads.

The effects of memory system latency and resource contentions on JVM performance were studied. It was observed that increasing number of processors or application threads leads to increase in L2 cache misses and cache-to-cache transfer ratio which results in increased memory latency. The work presented in [27] studied scalability of Specjbb2005 on CMP architectures (up to 64 threads) with number of sockets, number of cores and cache size. It was observed that Specjbb2005 performance is heavily tied to cache and memory performance and having a DRAM cache can achieve up to 40% performance improvement.

In [43] scalability of a collection of commercial workloads on a multi-core system, SUNFireT2000 was reported. The workloads include SAP-SD, IBM Trade, Specjbb2005, SPEC SDET, Dbench and Tbench. Apart from application throughput, CPI and memory performance (miss rates for Data TLB, L1 and L2 caches) was studied with increasing number of cores and increasing number of hardware threads per core. The applications mostly exhibit a close to linear scaling with number of cores and about 50% to 70% scaling efficiency with number of hardware-threads per core. The multi-core architecture mitigated the effect of latencies due to inter-thread resource contentions. For Trade, colocating similar tasks on a set of dedicated cores was shown to improve performance. Software design (both middleware and application code) poses another dimension affecting workload scalability.

In particular, lock contentions, garbage collection, application server configuration, shared data structures can limit scalability. Techniques for improving scalability and performance of application servers include session/transaction affinity, workload segmentation into service classes, request aggregation into batches, efficient management (creation and reuse) of connection pools at different layers like web-container and database, and intelligent caching of web pages. The work presented in [23] provides a comprehensive examination of techniques for improving scalability and performance of IBM WAS (Websphere Application Server) Identification of lock contention is tied to solution patterns used in application development.

In a recent work [24] the authors studied scalability of three server-side Web applications on a chip multiprocessor by reducing lock contention. Three prominent solution patterns were proposed to reduce lock contention: Concurrent Hashmap, Double-checked locking and Read-write lock. Incorporation of these patterns resulted in improvement in throughput. The work presented in [35] studied emerging Web 2.0 workloads from applications like Mashups, Wikis, Blogs, and the specific scalability issues resulting from participatory nature of these workloads on multi-core architectures (IBM POWER6 and SUN Niagara 2). These workloads are characterized by frequent interaction with the persistence layer resulting in lock contention.

IX. Conclusion and Future Work

Java applications face significant scalability and performance challenges in the context of massive multi-core systems mostly because of the complexity of their software stacks. Indeed, the more software layer used between an application and a system, the more likely scalability bottlenecks might be hit. After giving an overview of a traditional software stack for a Java-enterprise workload, this paper proposed the evaluation of three enterprise-grade Java-based application with an eye to determine whether or not a common contention scheme could be defined.

Since CGS behaviour was identical on both POWER7 and x86 architectures, we expect these findings to be valid for other hardware configuration. Two observations can be made. The first observation is that the scalability bottlenecks of the considered applications are mostly lock contentions and

were found in most of the layers of the software stack: the application layer, the middleware layer, the pure JCLs layer, and the native JCLs layer. The second observation is that none of the discovered contentions is shared across the workloads; in other words, each new considered workload brought its own share of bottlenecks. Although the first observation is relatively mundane [24], [28], [38], [7], [16], [33], the second observation is rather uncanny – given the complexity of each workload and the amount of code involved, we expected at least one shared bottleneck – and leads us to believe that the evaluation of a fourth Java-based application is likely to present yet another bottleneck source.

As a result, no common bottleneck for the class of Java-based enterprise applications can be identified from our evaluation. A well defined, streamlined evaluation methodology as the one defined in section II-C is still mandatory to evaluate scalability bottlenecks in a complete Java-based software stack. However, considering the class of Java-based enterprise application in its entirety is probably too coarse grained; further studies of various subclasses (e.g. the BRMS subclass, the PDF generation subclass, or the chart generation subclass) could cast light on potential trends.

Future work involves extending this study to enterprise applications in analytics, business intelligence and business process management area and identifying fundamental issues in scalability. At the same time, we feed the gained knowledge back to the owners of the various stack layers to get the optimizations implemented in the next generation of products. Our ultimate goal is to achieve near-perfect scalability for Java enterprise workloads, study the impact of low-level operating system and micro-architectural details on performance, and compare the results to scientific applications known to scale.

REFERENCES

[1] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *Proc. of the ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 739–753, 2010.

[2] E. A. Altman, M. Arnold, R. Bordawekar, R. M. Delmonico, N. Mitchell, and P. F. Sweeney. Observations on tuning a java enterprise application for performance and scalability. *IBM Journal of Research and Development*, 54(5):2, 2010.

[3] Apache Fundation. Apache DayTrader Benchmark Sample. http://cwiki.apache.org/GMOxDOC20/daytrader.html.

[4] Apache Fundation. Apache FOP. http://xmlgraphics.apache.org/fop/.

[5] Apache Fundation. Apache PDFBox. http://pdfbox.apache.org/.

[6] Apache Fundation. BCEL Manual at The Apache Jakarta Project. http://jakarta.apache.org/bcel/manual.html.

[7] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *ISCA*, pages 3–14, 1998.

[8] BFO. Big Faceless Java Graph library. http://bfo.com/products/graph/.

[9] BFO. Big Faceless Java PDF Library. http://bfo.com/products/pdf/.

[10] Z. Cao, W. Huang, and J. M. Chang. A study of java virtual machine scalability issues on smp systems. In *International Symposium on Workload Characterization (IISWC)*, 2005.

[11] P. De, V. Mann, and U. Mittaly. Handling os jitter on multicore multithreaded systems. In *IPDPS*, pages 1–12, 2009.

[12] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: a case study of building efficient, scalable Jvms. *IBM Journal of Research and Development*, 39(1), Jan 2000.

[13] EJ Technologies. The JProfiler Tool. http://www.ej-technologies.com/products/jprofiler/overview.html.

[14] C. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[15] GNUPDF. GNU JPDF. http://gnujpdf.sourceforge.net/.

[16] W. Huang, J. Lin, Z. Zhang, and J. M. Chang. Performance characterization of java applications on smt processors. In *ISPASS*, pages 102–111, 2005.

[17] IBM. IBM Cognos Business Intelligence. http://www-01.ibm.com/software/analytics/cognos/business-intelligence/.

[18] IBM. IBM Websphere ILOG Business Rules Management System. http://www-01.ibm.com/software/websphere/products/business-rule-management/.

[19] IBM. IBM Whole-system Analysis of Idle Time (WAIT). http://researcher.ibm.com/researcher/view_project.php?id=1332.

[20] IBM. Java Technology, IBM style: Garbage collection policies, Part 1. http://www.ibm.com/developerworks/java/library/j-ibmjava2/.

[21] IBM. JLM – Java Lock Monitor. http://perfinsp.sourceforge.net/jlm.html.

[22] IBM. Tprof - time profiler. http://perfinsp.sourceforge.net/tprof.html.

[23] IBM. Websphere application server v6 scalability and performance handbook (ibm redbook). http://www.redbooks.ibm.com/redbooks/pdfs/sg246392.pdf.

[24] K. Ishizaki, T. Nakatani, and S. Daijavad. Analyzing and improving performance scalability of commercial server workloads on a chip multiprocessor. In *International Symposium on Workload Characterization (IISWC)*, 2009.

[25] ISO. ISO 32000-1:2008 - Document management - Portable document format. http://www.iso.org/iso/catalogue_detail.htm?csnumber=51502.

[26] iText. iText PDF Generation Library. http://itextpdf.com/.

[27] R. Iyer, M. Bhat, L. Zhao, R. Illikkal, S. Makineni, M. Jones, K. Shiv, and D. Newell. Exploring small-scale and large-scale cmp architectures for commercial java servers. In *International Symposium on Workload Characterization (IISWC)*, 2006.

[28] J. Jann, R. S. Burugula, N. Dubey, and P. Pattnaik. End-to-end performance of commercial applications in the face of changing hardware. *Operating Systems Review*, 42(1):13–20, 2008.

[29] jCraft. JZlib - Pure Java Zlib. http://www.jcraft.com/jzlib/.

[30] jFree. JFreeChart. http://www.jfree.org/jfreechart/.

[31] jFree. jFreeReport. http://www.object-refinery.com/jfreereport/.

[32] R. Kalla and B. Sinharoy. POWER7:IBM's next generation power microprocessor. In *IEEE Symp. High-Performance Chips (Hot Chips 21)*, 2009.

[33] M. Karlsson, K. E. Moore, E. Hagersten, and D. A. Wood. Memory system behavior of java-based middleware. In *HPCA*, pages 217–228, 2003.

[34] T. Li, D. P. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPOPP*, pages 65–74, 2009.

[35] M. Ohara, P. Nagpurkar, Y. Ueda, and K. Ishizaki. The data-centricity of web 2.0 workloads and its impact on server performance. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[36] OW2. The Java ASM library. http://asm.ow2.org/.

[37] S. Seelam, Y. Liu, P. Dube, M. Ito, D. Binay, M. Dawson, P. Nagaraja, G. Johnson, L. Fong, M. Hack, X. Meng, Y. Gao, and L. Zhang. Experiences in building and scaling an enterprise application on multicore systems. *Concurrency and Computation: Practice and Experience*, 24(2):111–123, 2012.

[38] Y. Shuf and I. M. Steiner. Characterizing a complex j2ee workload: A comprehensive analysis and opportunities for optimizations. In *ISPASS*, pages 44–53, 2007.

[39] SPEC. SPECjvm2008 Java Virtual Machine Benchmark. http://www.spec.org/jvm2008/.

[40] L. Stepanian, A. D. Brown, A. Kielstra, G. Koblents, and K. Stoodley. Inlining java native calls at runtime. In *VEE*, pages 121–131, 2005.

[41] Sun Microsystems. Improving java application performance and scalability by reducing garbage collection times and sizing memory using jdk 1.4.1. http://developers.sun.com/mobility/midp/articles/garbagecollection2/.

[42] The Linux Kernel Organization. The Linux Kernel. http://kernel.org/.

[43] J. H. Tseng, H. Yu, S. Nagar, N. Dubey, H. Franke, P. Pattnaik, H. Inoue, and T. Nakatani. Performance studies of commercial workloads on a multi-core system. In *International Symposium on Workload Characterization (IISWC)*, 2007.

[44] X10. X10: Performance and productivity at scale. http://x10-lang.org.

[45] L.-J. Zhang, J. Zhang, and H. Cai. *Service Computing*. Tsinghua Univerity and Springer, 2007.