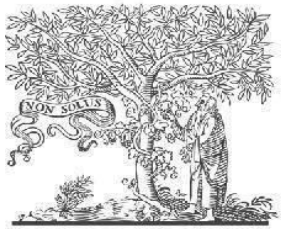
International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

COPY RIGHT



ELSEVIER SSRN

IJIEMR Transactions, online available on 30th Nov 2024. Link

https://ijiemr.org/downloads.php?vol=Volume-13&issue= Issue11

DOI:10.48047/IJIEMR/V13/ISSUE11/17

Title: " IMPROVING MIDDLEWARE APPLICATION PERFORMANCE WITH JVM TUNING AND THREAD DUMP ANALYSIS"

Volume 13, ISSUE 11, Pages: 167- 178

Paper Authors
Venkat Marella

USE THIS BARCODE TO ACCESS YOUR ONLINE PAPER

To Secure Your Paper as Per UGC Guidelines We Are Providing A ElectronicBar code

International Journal for Innovative Engineering and Management Research
PEER REVIEWED OPEN ACCESS INTERNATIONAL JOURNAL
www.ijiemr.org

IMPROVING MIDDLEWARE APPLICATION PERFORMANCE WITH JVM TUNING AND THREAD DUMP ANALYSIS

Venkat Marella

Independent Researcher,USA.

Abstract

Java's speed is now on par with C/C++ thanks to recent improvements in Java class libraries and Just-in-Time (JIT) compilation methods. An appealing objective is to make the Java Virtual Machine (JVM) "cluster-aware" in order to fully utilize Java's multithreading feature on clusters. This would allow a collection of JVMs operating on dispersed nodes to function as a single, more potent JVM, enabling true parallel execution of a multithreaded Java application. However, there hasn't been much research done on using DJVMs to cluster real-world web applications with workloads from commercial servers. In this article, a novel generic clustering method based on DJVMs that encourages global object sharing and user transparency for web application servers is presented. The optimization methods recently used for the Just-In-Time compilers included in the IBM® Developer Kit for JavaTM and the J9 Java virtual machine standard are described in this document. It mostly concentrates on the improvements that enhanced the performance of the middleware and servers. Virtual machines (VMs) and just-in-time (JIT) compilers face a number of performance issues when it comes to large server and middleware applications built in the Java programming language; we need to handle both steady-state performance and startup time. We outline 12 enhancements that have been made to IBM products in this article because they enhance the scalability and performance of these kinds of applications. For instance, these optimizations lower the cost of object creation, synchronization, and other frequently used Java class library calls. We also outline methods, including recompilation strategies, to deal with server start-up time. According to the experimental findings, the modifications we go over in this article may increase the performance of programs like SPECjbb2000 and SPECjAppServer2002 by up to 10% to 15%.

Keywords: - JVMs Running, C/C++, Large server, SPECjAppServer2002, Developer Kit, J9 Java, 12 optimizations, SPECjbb2000, Java Programming Language, Server Workloads, DJVMs, Middleware Applications.

I. INTRODUCTION

The need for computing power is rising annually due to the continuing Industrial Revolutions 4.0 and 5.0 as well as the increased prevalence of Internet access in digital societies. Global commercial Internet service providers (like Netflix and LinkedIn) and IoT-related platforms (like Bosch IoT Suite, [1], and Samsung SmartThings Cloud), as well as government organizations that offer informational services to their citizens nationwide (like e-Health) and businesses (like e-Banking), face a unique challenge in maintaining the availability of digital services they offer with steadily rising user volumes. These problems are associated with the so-called scalability of computing devices, which is defined as ensuring a steady level of

quantifiable quality indicators of a system's usage as the number of users or demands for it increases. Naturally, a higher need for processing power results from increased requests per unit of time [1, 2], which may be readily supplied by expanding the computing node's CPU cores or RAM. This is referred to as scaling in the vertical direction, and while it seems to be the most straightforward [2, 3], it is constrained by the current state of computer architecture. Horizontal scaling, on the other hand, entails expanding a computer system's number of computational nodes [3]. Naturally, this is much more complex because it necessitates extra overhead for system orchestration, which includes provisioning computing node locations, deploying successive instances of each service on them, keeping an eye on the load, and, lastly, a particular method for developing software that has minimal overhead for deployment and operation. Two factors make this especially crucial [3, 4]. The first is to facilitate the growth of server-based systems installed in sizable data centers, which provide their services to other customers, including Internet of Things devices. The second is to boost the processing capability of edge computing systems, because individual computer nodes—which often support or are IoT devices themselves—have a restricted capacity [4].

Research initiatives like DJVM have shown promise in terms of scientific standards. Nevertheless, it is more difficult and has not been well investigated to cluster real-world business applications with servers operating on top of DJVMs [4, 5]. By displaying the performance outcomes of running Apache Tomcat on the JESSICA2 DJVM, this study fills the gap. The runtime characteristics of Tomcat, a typical object-based server, include high read/write ratios, significant object sharing via the collections framework, fine-grained irregular object access patterns, and heavy threading [4, 5]. We test them on a variety of web apps and explore their possible effects on the DJVM performance. We demonstrate that the DJVM technique can scale out transparently and as effectively as conventional clustering for I/O-centric applications [5]. Our distributed shared heap architecture provides global cache hits, which significantly improve performance for cache-centric workloads. A taxonomy of current web application clustering technologies with their common flaws revealed, a description of DJVM-level overheads for applications on servers, and recommendations for the design and optimization of the next-generation DJVM are some of the work's other contributions [5, 6].

A Java Virtual Machine (JVM) is the platform on which Java applications operate. Multithreaded Java programs cannot be executed with good performance with the existing JVM technology. The primary focus of standard JVMs, such Sun JVM, IBM's JVM, and others, is on fast native execution on a single node [6, 7]. These systems, however, are unable to expand to huge environments like clusters. Our research focuses on distributed Java virtual machines (DJVMs), which may provide multithreaded Java applications running on clusters a high-performance execution architecture [8]. The study will promote Java parallel programming to boost cluster software production. The DJVM is systems that are distributed made up of many "cluster aware" JVMs operating on clusters that cooperate to form a single, more potent JVM. Java threads produced inside a single application may execute concurrently on many cluster nodes when using a DJVM. The DJVM system is completely consistent with the Java language specification and offers Java applications all the virtual machine functions

of a normal JVM [8, 9]. In contrast to the stand-alone JVM, which is limited by the number of processors on a single server, the DJVM technique enables the current multithreaded Java applications to fully leverage the processing capabilities available in all participating nodes [9, 10].

We think that a DJVM requires the following capabilities in order to take use of clusters' increased computing power and to make the system scalable [10, 11].

- **High-speed native execution:** A DJVM should include a Just-in-Time (JIT) compiler to enhance the performance of Java programs. Many times quicker than an interpreter, a 2 JIT compiler dynamically converts Java bytecode into native code as it is being executed [12, 13]. The laborious interpretation layer that sits between the program and the hardware is eliminated by JIT compilation. The produced native code may directly modify the hardware machine registers [14]. Additionally, in JIT compilation mode, a broader range of computer code may be bundled together for code optimization.
- **High parallelism:** A large-scale parallel hardware platform is offered by the cluster. The DJVM should include built-in cluster-wide multithreading to enable the high-performance operation of multithreaded Java programs on such a platform [15]. Large-scale parallel processing is supported by the functionality, which allows Java threads to be effectively transferred to various cluster nodes [15, 16].
- **Resource utilization and load balancing:** For large-scale multithreaded Java programs to achieve scaled performance on clusters, load balancing and efficient resource use are crucial [17]. Pre-emptive thread migration, which permits a thread to migrate across computers while it is being executed, should be supported by the system. In this manner, additional idle computers may dynamically join a program's execution to increase performance on the fly, and more sophisticated thread scheduling policies can be built on migration support for balancing the system burden [17].
- **Efficient distributed object access:** Every thread in a conventional Java virtual machine uses the same heap to access data [18]. All distributed Java threads should have access to a distributed shared heap in order to offer the same support in a DJVM [18, 19]. Investing in optimization strategies is necessary to lower object sharing overheads.
- **Programmability:** DJVM should not need the introduction of new APIs or language changes. Multithreaded Java applications may be written by the programmer in the standard way [19]. Without any changes, an existing application that can run on a single-node JVM can run on a DJVM. To guarantee programmability, the DJVM should have a single-system visual illusion.

However, there are several additional characteristics of server and middleware programs that adversely affect their performance, even beyond this specific difficulty [19, 20]. In this article, we outline 12 distinct improvements and optimizations that have been added to IBM products to improve these applications' performance [20, 21]. These characteristics include, for instance, optimising Java class libraries to increase the speed of, say, calculating transaction time-stamps, and improving synchronization to decrease recurrent locking and unlocking on the same thing object to enhance scalability [21, 22].

However, there are other crucial performance factors outside steady-state performance. For instance, in order to prevent expensive interruptions in service, application servers need to be able to start up rapidly when a computer is restarted. For application developers, who often need to restart the application server as part of the standard edit-compile-debug development cycle, a quick start-up time also significantly increases productivity. In this study, we demonstrate how several recompilation methods may be used by the JIT compiler to greatly reduce server start-up time [14, 16].

With a primary emphasis on improvements that enhanced server and middleware performance, this paper details newly implemented optimization methods used by the Just-In-Time compilers included in the IBM® Developer Kit for Java and the J9 Java virtual computer standard.

Performance problems were common in early versions of Java because it employs more high-level features than, say, C. For instance, garbage collection [16,17], which significantly reduces associated issues such memory leaks and simplifies the management of memory, also consumes a significant amount of computation time. The situation in question was improved to the point that performance issues, as compared to native code solutions, largely arise for extreme situations with unique needs, such hard real time, thanks to developments in garbage collection techniques and the advent of a Just-in-Time compiler. By automatically allocation and disposal of memory, garbage collection frees the programmer from manual memory management so they may concentrate more on the real core functionality [18]. To find wasted memory, free it, and make it useful again for the application's subsequent execution, nearly every version of Java virtual machines include at least one garbage collection technique [19]. John McCarthy first used the phrase when he created a reclamation cycle in LISP to release memory shortly after an allocation [19, 20].

1.1 Java Coding Observations

Our teams come into contact with a lot of Java code that has been developed by our customers as a result of fixing their bugs [20]. In this part, we outline what we consider to be three significant findings about the code being used that directly affect Java application performance:

1) A shift away from Java code translation by Java and toward bytecode creation,
2) The use of finally blockages more often, [20, 21], and
3) The common use of exceptions.

1.1.1 Bytecode Generation

The use of bytecode generating other than Java is growing. These include more advanced compilers like extended stylesheet language transformation (XSLT) compilers, as well as more basic utilities like Java Server Pages (JSP) servlet generators [23]. These tools may produce bytecode streams that affect virtual machine performance generally and specifically when interacting with JITs. Improvements in JIT implementations may help with some of these interactions, while careful bytecode generator implementation is needed for others.

1.1.2 Finally Blocks

Finally blocks provide Java programmers the ability to ensure that an action is carried out whether control exits a method via an exception or a standard return route. This approach is used, for example, in middleware applications to ensure that a tracing function is called in the event that it is required to record that the method has finished running [23]. The majority of the method's code is contained inside a try area in this coding design, and the tracing code is positioned in a finally block for that try section [22]. This structure ensures that the tracing information is consistent.

1.1.3 Exceptions

We continue to come across programs where exceptions are thrown in frequently performed routes, even though there is enough evidence of the hidden cost of handling exceptions. Even while JITs work hard to reduce the time lag between throwing an exception and the code that handles for that exception starting to run, the latency is still substantial and an amount of orders of magnitude longer than when executing a branch, for instance [22]. Because every frame has to be searched for a catcher and locked objects that need to be opened, throwing an exception that will be caught greater in the stack might be very costly. A contrived catch block that unlocked the object and then free-throws the exception is often used to solve the latter issue [21]. Processing the exception will be much more costly if the "real" catcher is located several frames above the frame where the exception was first thrown because there are many objects to unlock in the intervening frames; this will require throwing the exception multiple times.

II. SERVER PERFORMANCE WORK

Eight areas where we have improved server application performance (either in the VM or in the JIT) are described in this section: String, new Instance [24]. System, Index Of (). System, Present Time Millis (). Using the Intel® SSE instructions, thread-local heap batch cleaning, lock coarsening, array copy (), and object allocation in lining. The SPECjbb2000 benchmark is specifically the focus of several of these improvements [23].

2.1 New Instance

The class Java.lang. An instance of the same type as the Java.Lang.Class object supplied as the argument to new Instance is returned by the New Instance method [24]. The class library's new Instance method invokes a native new Instance Impl techniques that performs three functions: first, it determines whether the class being instantiated has a default constructor; second, it determines whether the caller can access this default constructor; [26]; and third, it invokes the constructor. There are many reasons why this technique is ineffective:

1) The pricey invocation of the native new Instance Impl function.
2) Looking for the constructor by default.
3) Walking the stack to make that the constructor is not being run at the moment may be necessary to confirm that it is visible in the caller's context.

The costly call-back from the native procedure into the Java constructor [23, 24].

- ✓ **String. Index Of ()**

There are many methods, including Boyer and Moore's approach, that carry out the semantics that the Java class library's String.indexOf () function specifies. Many of these computations work in two stages: they quickly cycle over the source string after first computing some meta-data depending on the target string [22]. Our examination of benchmarks like SPECjbb2000 showed that short constant pattern strings are often used to trigger the index of. Because the compiler can statically generate the meta-data related to the pattern string, [25], Boyer and Moore's approach is perfect in this scenario. Only the quick phase has to be carried out in response to the actual invocation of the index of.

- ✓ **System. Current Time Millis ()**

The system for Java. Transaction-based server programs that continually generate time stamps, like SPECjbb2000, often utilize the Current Time Millis () function. This approach is costly because:

A costly call to a native method is made from JIT-ted code [23].

The method returns a lengthy value that may either be accessed more slowly via the stack or stored in a register, which increases register pressure.

- ✓ **System. Array copy ()**

System is one of the most often utilized intrinsics in Java middleware programs. [2] Array copy [2]. The ideal code order for this technique varies depending on the size of the array that has to be duplicated and is platform-specific.

- ✓ **Object Allocation IN lining**

JITs may now use a Thread Local Heap (TLH) to perform typical object allocations thanks to features provided in almost every version of the IBM Developer Kits. A TLH is a little area of the heap that is momentarily designated for a certain thread's exclusive usage. The area returns to being a normal portion of the heap when there are no more allocations available in a TLH [11].

- ✓ **Lock Coarsening**

It is not unusual to synthesize a block of code that has many lock and unlock operations performed serially (i.e., not in nested manner) to the same object since the JIT aggressive in lines invocations while building a method [11]. Inlining synchronized functions into the code being built is what causes these lock and unlock procedures. SPECjbb2000 analysis revealed a technique that, depending on the route taken, performed six or seven iterative lock and unlock operation on the same object.

- ✓ **Thread-local Heap Batch Clearing**

To lessen heap lock contention, the IBM JITs use a thread-local heap. Instead of initializing those values for each every object allocation, it may be more efficient on certain processors to initialize the thread-local heap's whole contents to zero when it is created. Specifically, a whole

cache line may be initialized to 0 at a time using the IBM PowerPC® architecture [11]. This batch clearing technique is only used by IBM JITs for CPUs with effective architectural assistance with initializing large memory blocks. The items taken from the local heap are initialized separately for each CPU.

✓ **Intel® SSE Instructions**

Streaming SIMD Extensions (SSE), a technique intended to speed up applications like 2D/3D graphics, image processing, and voice recognition, were added with Intel's Pentium® III CPU. A file containing eight 128-bit XMM registers is included. According to the SSE architecture, four single-precision floating-point values may be stored in each XMM register. Two double-precision floating-point values (or four single-precision values) may be stored in each register thanks to a further expansion (SSE2) that was added to the Xeon™ and Pentium 4 CPUs [16]. The extensions include a comprehensive collection of instructions for modifying the data in XMM registers as well as for adding and removing data from them.

III. MIDDLEWARE PERFORMANCE WORK

In order to enhance the functionality of middleware programs such as SPECjAppServer2002, we have lately concentrated on four areas, which we outline in this section: increasing interface invocations through polymorphic inline caches, decreasing application server start-up time through recompilation techniques, identifying instances in which 64-bit long variables are being used to (inefficiently) carry out unsigned 32-bit computations, and rearranging code to minimize branch mispredictions and instruction cache misses [12, 13].

3.1 Application Server Start-up Time

Since the application server is often launched at least once during each edit-compile-debug cycle, its start-up time is crucial for a number of reasons, including speedy recovery from server failures and reduced development time [13].

3.2 Polymorphic Inline Caches

Java invocations may be polymorphic, meaning that the runtime type of the receive object determines the actual target when a method is called [13, 14]. Programmers gain from this feature as it makes it possible to create code that is understandable, reusable, and maintainable. However, this advantage comes at the often severe expense of requiring the system to decode the receiver object's type before calling the proper target for each polymorphic invocation [15]. Because a class might implement many interfaces in Java, it can be especially costly to determine the suitable target for an invocation via an interface method.

3.3 Unsigned Arithmetic for Cryptography Applications

Since SSL (Secure Sockets Layer) was introduced and used in Java-coded middleware application servers [16], the security protocol's speed has become just as crucial as that of conventionally insecure transaction processes [17]. A secure application's speed might be considerably hampered by this secure layer overhead as compared to a non-secure version.

```
void aMethod(...) {
   if (_trace.entryTracing()) {
      _trace.entry("aMethod");
      // log arg info
   }

   // do the work of aMethod

   if (_trace.exitTracing()) {
      _trace.exit("aMethod");
       // log effect
       // log return value info
   }
}
```

3.4 Code Reordering

Many middleware applications include methods that incorporate tracing code that conditionally runs at the start and/or finish of the function to record the application's execution as a debugging tool [19]. This function is seldom used in a production environment and is usually run while troubleshooting the program since the trace information is usually rather large.

IV. RESULTS

In this part, we describe the advantages we have seen in a set of performance-related benchmarks for middleware and server applications [20]. Because it is not always feasible or acceptable to maintain code to selectively activate or disable certain features, the improvements we report were gathered throughout the course of the most recent product development cycle rather than at a single point in time. When such code does not increase serviceability in a production JIT compiler, it is removed after a testing cycle to make code maintenance easier [21, 22].

4.1 SPECjbb2000

Without the use of an application server or database layer, the SPECjbb2000 benchmark simulates a pure server application. The database and logic for a company's order, inventory, delivery, and payment management are encoded by the benchmark. Although it is not exactly analogous to the TPC-C database workload, it is influenced by it [23]. By gradually increasing the number of operational warehouses, the benchmark gradually raises its workload to measure the throughput provided by a certain system. It starts at one warehouse and ends at two hundred and nine warehouses, where N is typically the number of processors in the machine [23]. Improvements to the final score, an aggregate metric that mainly monitors the average of the throughputs attained at warehouses P through 2P—where P is the warehouse where the peak throughput was observed (often P = N)—are the findings shown below.

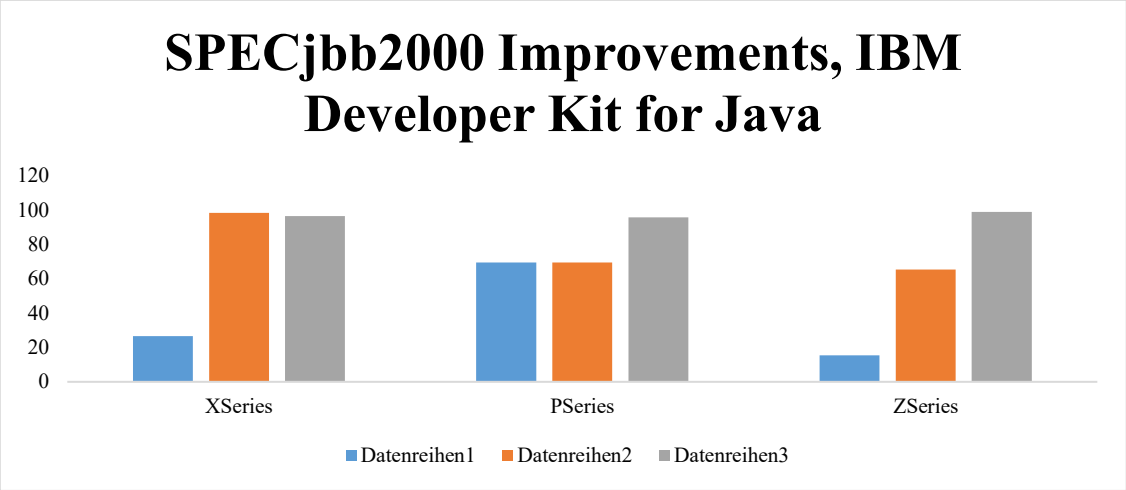


Fig. 1 Enhancements to the IBM Developer Kit for Java, SPECjbb2000. [22]

4.2 SPECjAppServer2002

A large-scale middleware application benchmark called SPECjAppServer2002 simulates every aspect of a Fortune-500 company's operations, from manufacturing to inventory and supply chain management to consumer ordering and billing [22, 21]. Both an application server and a database layer are exercised in the benchmark [22], and each tier may be spread over several servers, run on the same system, or run on separate machines.

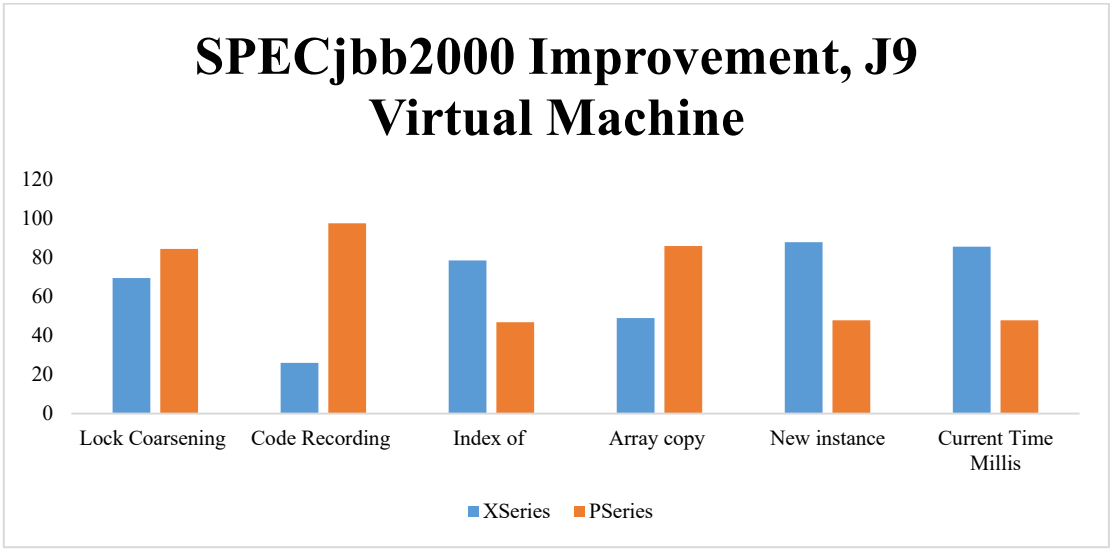


Fig. 2 Enhancement of SPECjbb2000, J9 virtual machine. [23, 24]

4.3 XML Parser

Our XML Parser benchmark is based on the Apache Xerces XML Parser for Java, which evaluates parser performance across various XML data set combinations [21]. Extensibility is one of the primary design objectives for the Xerces parser; the implementation of the parser mostly depends on abstractions and method invocations via interface classes.

We used the IBM Application Server WebSphere product as the foundation for our application server start-up performance investigation, and we provide the improvement levels that may be

attained by using multi-level optimization recompilation methodologies [18, 19]. Since a large amount of class loading happens at server start-up, we demonstrate that lowering the class loading cost by disabling class verification may result in a considerable increase in performance. The percentage gains in relation to the interpreter's time to boot the application on the server to its ready state are shown in Figure 5 [20]. The IBM WebSphere Application Server software was used to run the J9 virtual machine on an xSeries 1.6 GHz Pentium 4 uniprocessor running Microsoft Windows 2000 in order to gather the findings.

V. CONCLUSION

In summary, the technologies examined here provide effective software development within a micro services architecture, which promotes the horizontal scalability of system components that need it. Each of the evaluated solutions represents a significant advance in the development of high-availability, global IT systems because horizontal scalability provides almost infinite options to satisfy the system demand.

In light of this, we propose a few design principles for DJVMs of the future. In order to prevent the cluster-wide locking overheads from counteracting the beneficial cache impact of global object sharing, it is first necessary to maintain high execution concurrency. We discover that the primary cause of remote locks is thread-safe Java collection classes, which are often used as containers for shared objects. Actually, the most recent Java concurrent utility package has led to the development of more scalable containers. However, in terms of performance, these concurrent data structures are not instantly transferable to cluster contexts.

We have made three fundamental contributions in this study. First, we made three observations regarding Java coding practices among our customers that directly impact the performance capabilities of JITs and VMs: bytecode generation, the more prevalent use of finally blocks and the continuing frequent use of exceptions. Second, we outlined 12 key enhancements and optimizations that our teams have recently created to enhance the functionality of middleware and server applications. Third, we provided findings demonstrating the advantages of strong implementations of these optimizations for a range of uses. These outcomes show how well the features we've added work and how much improvement is really possible for strong implementations in a high-performance production JIT.

VI. REFERENCES

[1] Zeichick. Tomcat, Eclipse named the most popular in SDTimes study.

[2] W. Fang, C. L. Wang, and F. C. M. Lau. Efficient global object space support for distributed JVM on cluster. In Proc. Int. Conf. on Parallel Processing, pages 371–378, British Columbia, Canada 2002.

[3] ObjectWeb. JMOB: Java Middleware Open Benchmarking. http://jmob.objectweb.org/. [12] B. Goetz. Threading lightly, Part 3: Sometimes it's best not to share - exploiting ThreadLocal to enhance scalability.

[4] D. Lea. Java concurrent utility package.

[5] D. J. Scales, and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In Proceedings of the sixteenth ACM symposium on Operating systems principles, Saint Malo, France, 1997.

[6] Y. Aridor, M. Factor, A. Teperman et al. Transparently obtaining scalability for Java applications on a cluster. Journal of Parallel and Distributed Computing, v.60 n.10, p.1159-1193, Oct 2000.

[7] Balakrishna, S.; Thirumaran, M. Semantic interoperability in IoT and big data for health care: A collaborative approach. In Handbook of Data Science Approaches for Biomedical Engineering; Elsevier: Amsterdam, The Netherlands, 2020; pp. 185–220.

[8] Rajput, D.S.; Gour, R. An IoT framework for healthcare monitoring systems. Int. J. Comput. Sci. Inf. Secur. 2016, 14, 451.

[9] Baranwal, T.; Nitika; Pateriya, P.K. Development of IoT based smart security and monitoring devices for agriculture. In Proceedings of the 2016 6th International Conference-Cloud System and Big Data Engineering (Confluence), Noida, India, 14–15 January 2016; pp. 597–602.

[10] Gunasekera, K.; Borrero, A.N.; Vasuian, F.; Bryceson, K.P. Experiences in building an IoT infrastructure for agriculture education. Procedia Comput. Sci. 2018, 135, 155–162.

[11] Ryu, M.; Yun, J.; Miao, T.; Ahn, I.Y.; Choi, S.C.; Kim, J. Design and implementation of a connected farm for smart farming system. In Proceedings of the 2015 IEEE SENSORS, Busan, Korea, 1–4 November 2015; pp. 1–4.

[12] Misbahuddin, S.; Zubairi, J.A.; Saggaf, A.; Basuni, J.; Sulaiman, A.; Al-Sofi, A. IoT based dynamic road traffic management for smart cities. In Proceedings of the 2015 12th International Conference on High-Capacity Optical Networks and Enabling/Emerging Technologies (HONET), Islamabad, Pakistan, 21–23 December 2015; pp. 1–5.

[13] Patti, E.; Acquaviva, A. IoT platform for Smart Cities: Requirements and implementation case studies. In Proceedings of the 2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a Better Tomorrow (RTSI), Bologna, Italy, 7–9 September 2016; pp. 1–6.

[14] Bauer, M.; Sanchez, L.; Song, J. IoT-enabled smart cities: Evolution and outlook. Sensors 2021, 21, 4511.

[15] Stolojescu-Crisan, C.; Crisan, C.; Butunoi, B.P. An IoT-based smart home automation system. Sensors 2021, 21, 3784.

[16] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In ACM Conference on Object- Oriented Programming Systems, Languages, and Applications, pages 108-123, Oct 1995.

[17] Kai Hwang and Zhiwei Xu. Scalable Parallel Computers: Technology, Architecture, Programming. McGraw-Hill, New York, 1998.

[18] A Lempel and J Ziv. On the complexity of finite sequences. IEEE Transactions on Information Theory, 22:75–81, 1976.

[19] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC), pages 229–239, New York, NY, 1986. ACM Press.

[20] David Gay, Rob Ennals, and Eric Brewer, Safe manual memory management, Proceedings of the 6th international symposium on Memory management (New York, NY, USA), ISMM '07, ACM, 2007, pp. 2–14.

[21] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney, Adaptive optimization in the jalapeno jvm, Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), OOPSLA '00, ACM, 2000, pp. 47–65.

[22] M. Bashiri and H. Karimi, An analytical comparison to heuristic and meta-heuristic solution methods for quadratic assignment problem, Computers and Industrial Engineering (CIE), 2010 40th International Conference on, july 2010, pp. 1 –6.

[23] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In Joint Symposium on Agent Systems and Applications / Mobile Agents, pages 16–28, 2000.

[24] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In Operating Systems Design and Implementation, pages 101–114, 1994.

[25] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. IBM Systems Journal, 39:175–193, 2000.