



Intel[®] Technology Journal

Managed Runtime Technologies

Enterprise Java Performance: Best Practices

Enterprise Java Performance: Best Practices

Kingsum Chow, Software and Solutions Group, Intel Corporation
Ricardo Morin, Software and Solutions Group, Intel Corporation
Kumar Shiv, Software and Solutions Group, Intel Corporation

Index words: Enterprise Applications, Application Servers, Java Performance, Java 2 Enterprise Edition, J2EE

ABSTRACT

This paper discusses best practices for maximizing the performance of enterprise Java* workloads. First, we introduce the importance of performance of enterprise Java applications. We then describe our top-down, data-driven, and closed-loop approach to characterize where the problems are. We examine the performance of the software/hardware stack, first from the system-level perspective (topology, I/O, network), then from the top software layer (application level), through the middle layer (Java Virtual Machine), and down to the platform layer (processor, memory). We conclude by summarizing our recommendations for attaining the best performance on enterprise Java applications.

INTRODUCTION

Managed runtime environments such as Java have proven to be a very attractive platform for developing and deploying enterprise applications. Accessible object orientation, programming safety, and automatic memory management features deliver a highly productive foundation for business application development. In addition, the platform independence offered by managed runtime environments provides unprecedented investment protection, which is appealing to Information Technology (IT) managers, as enterprise applications tend to have a long life span.

Advanced Just-In-Time (JIT) compilation, memory management, and garbage collection technologies have effectively addressed initial concerns raised about the poor performance of Java-based applications. Today's Java Virtual Machines (JVM*) take full advantage of a variety of target platforms, and keep up to date with the performance of the latest hardware and operating system advances as they evolve over time.

As Java [1] gained popularity in the development of server-based applications, standardized, robust, and scalable application support frameworks became a must. Enter Java 2 Enterprise Edition (J2EE) [2], a comprehensive specification for application servers, a class of system software designed to relieve application developers from creating and re-creating the “plumbing” necessary to support enterprise applications, including component models and life-cycles, object models, database access, security, transactional integrity, and safe multi-threading.

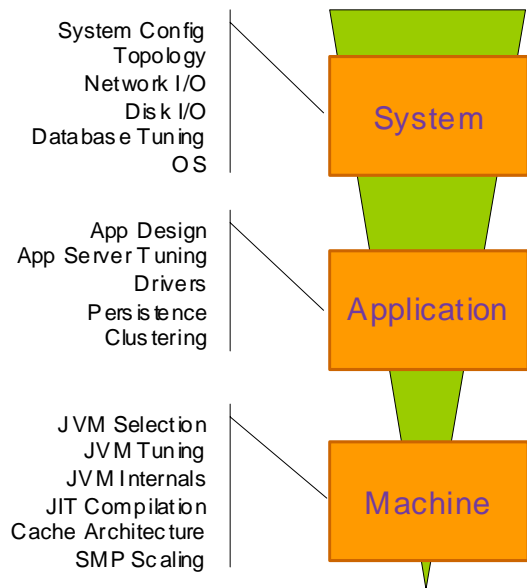


Figure 1: Performance optimization considerations at the three levels of the top-down stack: system-level, application-level, and machine-level

Since the emergence of J2EE, application servers have grown to become important IT infrastructure components of many enterprises [3]. They support complex, multi-tier configurations with well-defined separation of functions

(user interface, business processing, and database access), often including multiple servers arranged in clustered configurations, as well as back-end relational database management systems and legacy applications, integrated in the overall design.

As applications move from development to production, performance becomes a critical life-cycle requirement. Applications must not only meet stringent performance requirements upon deployment, but they must be able to gracefully scale with varying usage patterns and increased demand. Performance optimization and management in this environment is a difficult task, as performance is affected by many interrelated elements.

In this paper, we describe an iterative, data-driven, top-down methodology and the tools needed to systematically optimize the performance of application-server-based applications. We also describe performance optimization considerations at the three levels of the top-down stack: system-level, application-level and machine-level (see Figure 1).

At the system level, we identify performance and scalability barriers such as input/output (I/O), operating system and database bottlenecks, and we discuss techniques to overcome those barriers. At the application level, we discuss application design considerations and application server tuning. At the machine level, we discuss JVM implementations and hardware-level performance considerations such as processor frequency, cache sizes, and multi-processor scaling.

Throughout the paper, we introduce several case studies to illustrate the application of the techniques presented.

APPLICATION SERVERS

Application servers provide a solid foundation for developing and deploying enterprise applications. They implement a large collection of Application Program Interfaces (API) and a set of capabilities specified in the J2EE suite of standards, which support the development of multi-tier applications.

Application-server-based applications are arranged in multi-tier configurations: client tier, Web interface tier, business tier, and enterprise information systems tier. The client tier represents the service requestors, and it is usually associated with the user interface. The Web interface tier provides services required to process Web-based forms and Web services, and it dynamically assembles the resulting HTML and/or XML. The business tier is used to implement computation, business processing, and business rules. The enterprise information systems tier includes persistence back-ends, based on relational databases and legacy applications such as mainframe-based information systems.

As depicted in Figure 2, application server functionality is organized around the concept of containers, which provide groupings of related functions, and are typically layered on top of the Java 2 Standard Edition (J2SE^{*}) platform, which includes a Java Virtual Machine (JVM) and the corresponding suite of APIs. For instance, the two application server-based containers are the Web container and the Enterprise JavaBeans (EJB) container. Web containers are used to support Web-based user interface components, such as Servlets and Java Server Pages (JSP). EJB containers are used to support business components, which include Session Beans, Entity Beans, and Message-driven Beans. Session Beans provide access to independent business components in two flavors: Stateful Session Beans, used when state information is required between service calls, and Stateless Session Beans, used when individual service calls are independent of each other and do not require state information to be preserved. Entity Beans provide persistence services through connectivity to relational databases. Message-driven Beans provide the ability to implement business components that take advantage of asynchronous messaging capabilities.

In addition to the core container APIs, application servers provide additional support to APIs such as naming and directory services (JNDI^{*}), database connectivity (JDBC^{*}), messaging (JMS^{*}), XML processing (JAXP^{*}), transactions (JTS^{*}), and connectivity to legacy systems (JCA^{*}).

Most application servers also provide the ability to transparently cluster multiple containers in order to enable fault tolerance and multi-node scalability.

To provide runtime support for this comprehensive set of functionality, application servers need to implement a number of key services, including state management, life-cycle management, thread pooling, transactions, security, persistence, fault tolerance, and load balancing.

Examples of commercial application servers include BEA's WebLogic^{*} [4], IBM's WebSphere^{*} [5] and Oracle's 9i AS^{*} [6]. In addition, there are a number of open source implementations, including JBoss^{*} [7] and JOnAS^{*} [8]. Additional information about J2EE^{*} and application servers can be reviewed in [9], [10], and [11].

^{*} Other brands and names are the property of their respective owners.

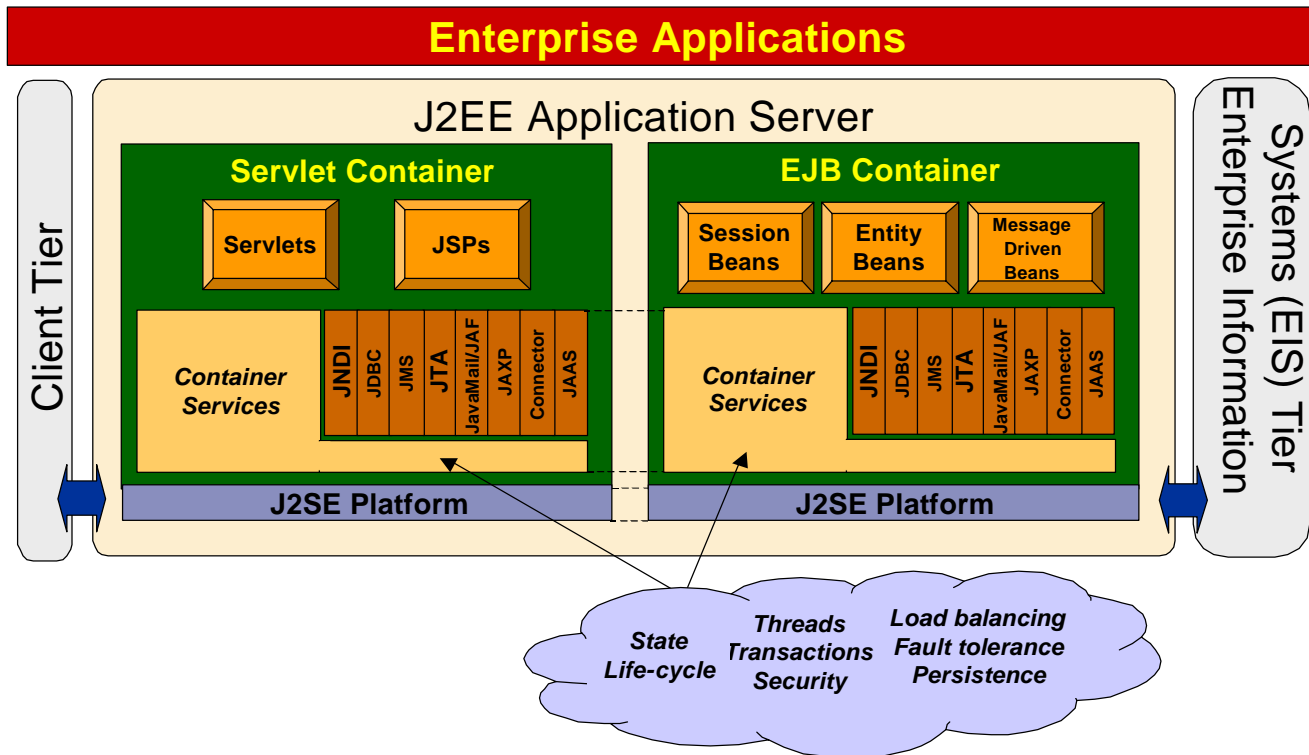


Figure 2: The two server-based containers are the Web and Enterprise JavaBeans (EJB) containers. Web containers support Web-based user interface components, such as Servlets and Java Server Pages (JSP). EJB containers support business components, which include Session Beans, Entity Beans and Message-driven Beans.

**PERFORMANCE TUNING
METHODOLOGY**

Application server configurations involve multiple computers interconnected over a network. Given the complexity involved, ensuring an adequate level of performance in this environment requires a systematic approach. There are many factors that may impact the overall performance and scalability of the system. Examples of these performance and scalability factors include application design decisions, efficiency of user-written application code, system topology, database configuration and tuning, disk and network input/output (I/O) activity, Operating System (OS) configuration, and application server resource throttling knobs.

The first and foremost element an application implementer needs to keep in mind to achieve the desired level of performance is ensuring that the application architecture follows solid design principles. A poorly designed application, in addition to being the source of many performance-related issues, will be difficult to maintain. This compounds the problem, as resolving performance

issues will often require that some code be re-structured and sometimes even partially re-written.

Once an enterprise application is ready for deployment, it is critical to establish a performance test environment that mimics production. This environment is then used to identify and remove performance and scalability barriers, using an iterative, data-driven, and top-down methodology.

A key consideration in the performance analysis process is selecting the workload. A good workload exhibits three fundamental attributes.

- First, the workload must be representative. It must provide adequate functional coverage, realistic implementation and usage patterns, and it must be relevant to the goal. The best workload is a controlled baseline of the application under study, configured as close as possible to production. It is highly desirable to fully populate the back-end database with realistic data in order to uncover data access bottlenecks, a common source of performance problems.

- Second, the workload must be measurable. It must have well-defined metrics and must exhibit a stable measurement period over which to gather performance statistics. Performance metrics for enterprise applications are usually defined in terms of throughput (number of operations per unit of time) and response time (amount of time that it takes to process individual transactions). Another commonly used metric is the number of simultaneous requests applied to the workload. This is often referred to as the injection rate. In many cases the total number of concurrent users achieves the same purpose.

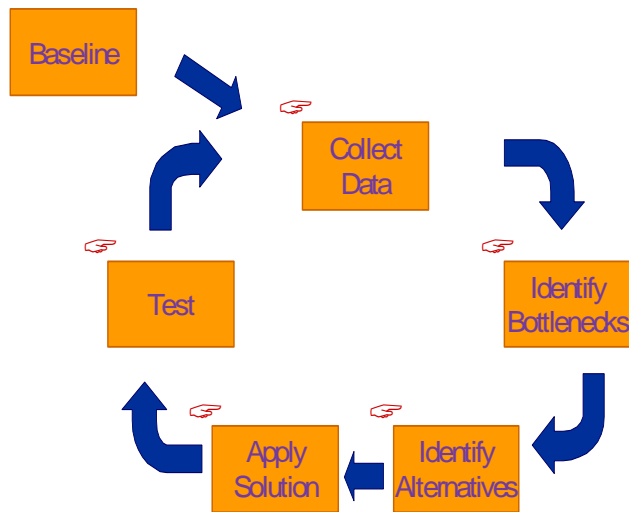


Figure 3: The iterative, data-driven, top-down process for performance tuning and optimizations

- Third, the workload must be repeatable. It needs to be consistent across tests, in order to be able to perform reliable data analyses and draw meaningful conclusions. In addition, the workload state needs to be the same at the start of each run. For example, if the workload adds data to the database, the database needs to be reinstated to the original state to avoid progressive performance degradation over successive runs. Variations in the primary metrics should not exceed a 5% margin across measurements.

Prior to engaging in performance tuning, it is important to establish a baseline to provide the basis for measuring performance improvements as the tuning process progresses. The baseline should be configured based on the estimated capacity needed to sustain the desired load, including network bandwidth and topology, processor memory sizes, disk capacity and physical database layout. In addition, the baseline configuration should incorporate basic initial configuration recommendations given by the application server, database server, JVM, and hardware platform vendors. These include: recommended tunable parameter settings, choice of database connectivity

(JDBC) drivers, and the appropriate level of product versions, service packs, and patches.

Part of the baselining process also includes defining performance goals for the system. Performance goals are usually defined in terms of desired throughput within certain response time constraints: for example, the system needs to be able to process 500 operations per second with 90% or more of the operations taking less than one second.

The steps in the iterative process, as illustrated in Figure 3, are as follows:

- Collect data: Use stress tests and performance-monitoring tools to capture performance data as the system is exercised.
- Identify bottlenecks: Analyze the collected data to identify performance bottlenecks.
- Identify alternatives: Identify, explore, and select alternatives to address the bottlenecks.
- Apply solution: Apply the proposed solution.
- Test: Evaluate the performance effect of the corresponding action.

Once a given bottleneck is addressed, additional bottlenecks may appear, so the process starts over again: performance data is collected and the cycle is initiated again, until the desired level of performance is attained. Two very important points to keep in mind during this process are first, let the available data drive performance improvement actions, and second, make sure only one performance improvement action is applied at a time.

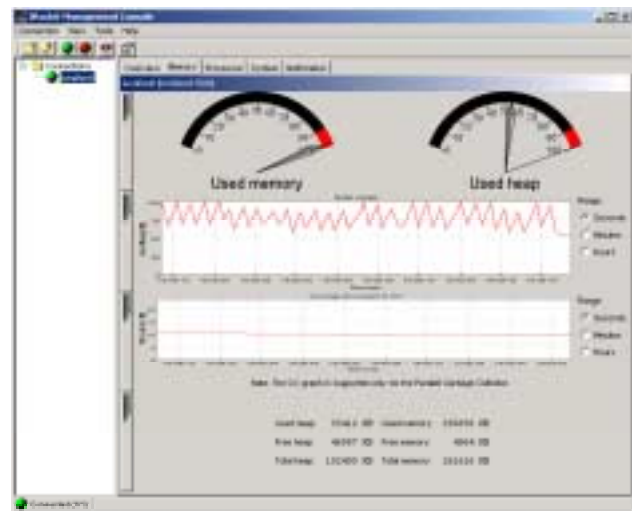


Figure 4: BEA WebLogic JRockit management console

As the quantity and variety of collected data can be overwhelming, and the bottlenecks can often come from

many interrelated sources, it is important to follow a top-down approach. At the top are system-level items such as disk subsystem configuration, network devices, and database configuration; in the middle are application-level items such as transaction configuration, persistence strategies, and JDBC drivers; and at the bottom are machine-level items such as JVM configuration, multi-processor configurations, and processor caches. The iterative process described above needs to be applied at each level of this hierarchy.

Having the right set of tools available is essential for supporting productive performance tuning activities. Performance tools fall under the following categories:

- Stress test tools. These provide the ability to script application scenarios and play them back, thereby simulating a large number of users stressing the application. Commercial examples of these types of tools are Mercury Interactive's LoadRunner* [12] and RADView's WebLoad* [13]; open-source examples include the Grinder* [14], Apache's JMeter* [15], and OpenSTA* [16].
- System monitoring tools. Use these to collect system-level resource utilization statistics such as CPU utilization (e.g., % processor time), disk I/O (e.g., % disk time, read/write queue lengths, I/O rates, latencies), network I/O (e.g., I/O rates, latencies). Examples of these tools are the Performance System Monitor from Microsoft's Management Console (known as perfmon*), and "sar/iostat" in the Linux environment.
- Application server monitoring tools. These tools gather and display key application server performance statistics such as queue depths, utilization of thread pools, and database connection pools. Examples of these tools include BEA's WebLogic* Console and IBM's WebSphere* Tivoli Performance Viewer.
- Database monitoring tools. These tools collect database performance metrics including cache hit ratio, disk operation characteristics (e.g., sorts rates, table scan rates), SQL response times, and database table activity. Examples of these tools include Oracle's 9i Performance Manager and the DB/2 Database System Monitor.
- Application profilers. These provide the ability to identify application-level hotspots and drill down to the code-level. Examples of these tools include the

Intel® VTune™ Performance Analyzer [17], Borland's Optimizeit* Suite [18], and Sitraka's JProbe* [19]. A new class of application response time profilers is emerging that is based on relatively modest intrusion levels, by using bytecode instrumentation. Examples of these include the Intel VTune Enterprise Analyzer [20] and Precise Software Solutions Precise/Indepth* for J2EE [21].

- JVM monitoring tools. Some JVMs provide the ability to monitor and report on key JVM utilization statistics such as Garbage Collection (GC) cycles and compilation/code optimization events. Examples of these tools include the "verbosegc" option, available in most JVMs, and the BEA WebLogic JRockit* [22] Console, depicted in Figure 4.

An important issue to keep in mind when using the above tools is that the measurement techniques employed introduce a certain level of intrusion into the system. In some cases, the intrusion level is so great that the application characteristics are altered to the extent that they make the measurements meaningless (i.e., Heisenberg problem). For example, tools that capture and build dynamic call graphs can have an impact of one or more orders of magnitude on application performance (i.e., 10-100X). The recommended approach is to only activate the appropriate set of tools based on the level the data analysis is focused on at the time. For example, for system-level tuning, it only makes sense to engage system monitoring tools, whereas application-level tuning may require the use of an application profiler.

Additional application tuning methodology information can be reviewed in [23].

SYSTEM-LEVEL PERFORMANCE

It is possible to identify two broad classes into which software can be bucketed, batch processing and interactive processing. For the former, the raw throughput, the amount of work done in a period of time, is the only real metric of interest. The time spent on any specific unit of work is not a significant consideration. For the latter class of software, the response time, the time taken for each unit of work, is very important, and in some cases it may be of higher importance than the throughput.

Architecting the system and application for good performance goes a long way towards making the rest of the performance optimization methodology more efficient. It is important to understand the type of the application

* Other brands and names are the property of their respective owners.

Intel® VTune™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

(batch or interactive) and to identify system hardware and software components that meet that goal [24].

Any system can be visualized as a network of components with transactions passing through them. In enterprise Java applications, such components could be viewed as hardware, software, or a combination of the two. For example, the network or disk sub-system is a hardware component, the application software is a software component, and the database server is a combination of hardware and software. Viewing the whole system as a network of components is useful in understanding the capacity requirements of system components.

Multi-processing, the capability of a system component to work on more than one request at the same time, plays a big role in the performance of most components. Two methods of multi-processing are pipelining and parallelism.

Pipelining is the concept of breaking down the required work into many parts. While one section of the component is working on one part of one transaction, other sections of the component can be working on other parts of other transactions, thereby maximizing use of system components. Pipelining is extensively used to increase throughput, but its effect on the time taken for an individual transaction is not a primary consideration.

Parallelism throws multiple resources at a task so that the task completes faster. Its primary effect is to reduce response time. Multi-threaded code is a way to achieve this in software. Hardware examples would include mirrored disks and multiple network cards.

Block diagrams that show the work-flow and identify the network of queues and parallel entities are very useful here. They are especially valuable in ensuring that sufficient capacity is designed into the system to meet the desired throughput and response time goals. Most systems typically use both multi-processing approaches.

Theoretically, the only system with no performance bottleneck is designed such that every component of the system exhibits the same performance behavior and has identical capacity. In practical terms, every system has a performance bottleneck.

At the system level, the goal is to ensure that the bottleneck is in the application code over which the developer has direct control, so that changes can be made that directly improve performance. If the bottleneck was elsewhere in the system, then even large-scale performance improvements in the developer's code may have only a slight effect on measured system performance.

Drawing a throughput curve can be very valuable in understanding system-level bottlenecks and helping identify potential solutions. Figure 5 shows a conceptual

diagram of a throughput curve, which plots system throughput, response time, and application server CPU utilization as functions of the injection rate, i.e., the rate of requests applied to the system.

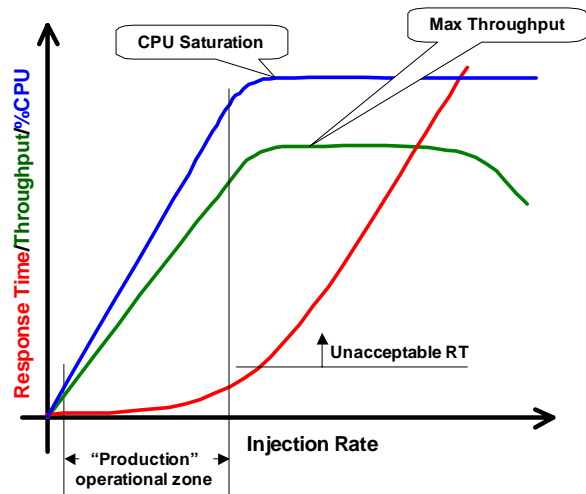


Figure 5: A conceptual throughput curve, which plots system throughput, response time and application server CPU utilization as functions of the injection rate, i.e., the rate of requests applied to the system

Through system-level tuning, the main goal should be to saturate the application server CPU (i.e., 90-100% utilization). Reaching maximum throughput without full saturation of the CPU is an indicator of a performance bottleneck such as I/O contention, over-synchronization, or incorrect thread pool configuration. Hitting a high response time metric with an injection rate well below CPU saturation indicates latency issues such as excessive disk I/O or improper database configuration.

Reaching application server CPU saturation indicates that there are no system-level bottlenecks outside of the application server. The throughput measured at this level would point out the maximum capacity the system has within the current application implementation and system configuration parameters. Further tuning may involve tweaking the application to address specific hotspots, adjusting garbage collection parameters, or adding application server nodes to a cluster.

Keep in mind that reaching CPU saturation is a goal for the performance tuning process, not an operational goal. An operational CPU utilization goal would be that there is sufficient capacity available to address usage surges.

Knowing the workload well is an important factor in the identification of the required capacity of many components. Some preliminary measurements and characterization can help. For instance, identifying the network bandwidth required for one unit of work will be

very useful in estimating the network capacity required when the desired system performance is N units of work.

Most components exhibit an exponential response time/throughput behavior. In other words, increasing the throughput will tend to increase the response time, and the higher the throughput, the faster the increase in response time. It is important to size these components such that the required throughput utilization for the component is relatively low to allow for the response time to be relatively small as well. This is especially important for network capacity, disk capacity, and the capacity of the data bus connecting processors to memory and I/O.

If the response time is an important aspect of the application, then low resource utilizations are particularly necessary; otherwise, higher utilizations will be acceptable. The precise thresholds that mark a utilization level as having hit a bottleneck depend on a variety of factors, and they are best identified through targeted experiments with test workloads.

System monitoring tools can be used to track system performance metrics, which can help find bottlenecks. In a multi-tiered system set-up where multiple computers are used, it is important to run these tools on all of the computers.

Key performance events that should be monitored include processor utilization, time spent in the kernel, interrupts, number of calls to the kernel (system calls), page faults, disk I/O, and network usage.

A key component of enterprise applications is a back-end relational database, as it provides essential persistence services, data retrieval capabilities for downstream systems, as well as support for querying and reporting applications. The back-end relational database is often a source of performance bottlenecks, because it manages large volumes of high latency disk I/O operations. It is, therefore, extremely important to pay special attention to the physical design and tuning of the database, to ensure acceptable levels of performance. Fundamental considerations include isolating log files to dedicated devices to reduce conflicts between the sequential nature of log operations and random access to data tables; adequately sizing the sort area memory size to minimize disk sort operations; allocating sufficient database cache memory (but avoiding swapping); carefully defining indexes such as indexing frequently used, highly selective keys, indexing foreign keys frequently used in joins, using full-text retrieval keys where appropriate; and using disk striping (e.g., RAID 1+0) to spread I/O operations and to avoid device contention.

Case Study 1: Database Tuning

The scenario described here was a performance issue related to database disk I/O, which is a common source of bottlenecks. In this case study, the system failed response-time requirements. Although throughput could be increased, response time increased as well. Also, the CPU was not fully utilized on the application server at maximum throughput and within response-time constraints. The supporting data included a high % disk time and long disk queues, high latencies (as seen in seconds per transfer), heavy log write activity, and excessive I/Os at some physical disks. The database used was Oracle, and Oracle statistics were helpful in pinpointing the source of the problem. In this case study, the data pointed to disk contention associated with the database log write operations. The solution was to isolate log files to dedicated devices to remove the perturbation of log write operations on other database activities, and to strip the log device to spread the I/O over multiple disks and reduce the associated latencies.

APPLICATION-LEVEL PERFORMANCE

Application design is one of the most important considerations for good performance. A well designed application will not only avoid many performance pitfalls from the start, but will be easier to maintain and modify during the performance testing phase of the development life-cycle.

Many J2EE application development best practices are well documented in design patterns [25] [26]. Design patterns provide a starting point for application design approaches that capture commonly encountered application functional requirements and usage scenarios. Several design patterns have positive performance implications, in addition to the associated maintainability and modularity benefits.

The following design patterns should be considered due to the clear performance advantages they provide:

- **Composite Entity.** This pattern provides mechanisms to implement coarse-grained entity beans that manage a set of subordinate persistent objects. It is used to limit the proliferation of entity beans and reduce the number of fine-grained remote calls.
- **Value Object.** This pattern assembles data requests into aggregated data objects to reduce remote calls to individual field get methods. It reduces the number of fine-grained remote calls and allows the transfer of more data with fewer remote calls.
- **Session Façades.** This pattern encapsulates business logic and data access using well-defined, coarse-grained, service-level interfaces to clients. It

eliminates the need for clients to access fine-grained business and data objects, thus reducing the number of remote calls. It is often used in combination with the Value Object pattern.

- **Service Locator.** This pattern encapsulates access to directory access through JNDI and provides caching of retrieved initial contexts and factory objects (e.g., EJB Homes). It reduces expensive accesses to JNDI by implementing caching strategies.
- **Value List Handler.** This pattern encapsulates access and traversal of database-generated lists of items. It improves performance by providing low-overhead list population mechanisms and implementing caching strategies.

In addition to design patterns, there are a number of programming practices that reduce performance bottlenecks, such as the following:

- Enterprise JavaBeans^{*} (EJB^{*}) homes and data sources should be cached to avoid repeated JNDI lookup of EJB objects and data source objects.
- Use of HTTP sessions should be minimized and used only for state that cannot realistically be kept on the client.
- Java Server Pages (JSP^{*}) create HTTP sessions by default. This should be overridden (i.e., session="false") when not needed, to prevent inefficient use of session resources.
- Database connections should be released when not needed as unreleased connections result in resource leakage problems.
- Unused stateful session beans should be removed explicitly, and appropriate idle timeout seconds should be set to control stateful bean life cycle to conserve scarce resources.

A strategy frequently used to improve the responsiveness and scalability of enterprise applications involves the use of asynchronous messaging, either through the use of message-driven beans or via JMS directly. Asynchronous messaging can be used to implement high latency business operations that do not require instantaneous processing, such as order requests or document submissions, thus increasing the responsiveness of the system [27]. Messaging can also be used to break down complex business operations in message processing pipelines, which can be parallelized by instantiating multiple message queue consumers. This enhances the scalability of the system by enabling multi-threading with minimal data sharing requirements.

While good practices are a good starting point for a high-performance system, they alone are not sufficient. The workload itself still plays an important factor in performance and it may demand a specific optimal application server configuration. Many parameters can be tuned to optimize for both response times and throughput, as reducing response time can often help increase the capacity for a further increase in throughput. However, when the response times are broken into sub-components, it is necessary to further tune the system so that the response times of key sub-components are optimized too.

Many of these tunable parameters are easily accessible from common application servers such as the BEA WebLogic server. The lists of parameters presented here to help improve performance are not exhaustive. They are merely good starting points to tune the performance for your enterprise Java applications. The list includes tuning key application server parameters and tuning key container parameters.

Tuning Key Application Server Parameters

Many application server parameters can be tuned to enable better sharing and interaction with virtual machines and operating systems. The following parameters should be considered for most applications.

- Platform-optimized socket multiplexers should be used to improve server performance for I/O scalability, because they overcome performance limitations of the blocking nature of Java I/O APIs prior to version 1.4. For example, when a performance pack is available from a vendor, it should be used.
- A thread pool size should be gradually increased until performance peaks. However, one should not make the number too big as a higher number may degrade performance. The optimal number is likely dependent on the workload and the performance metrics.
- An application server may support the notion of multiple queues for transactions, e.g., by allowing the application developer to choose different values of thread pool sizes for those queues. One may find a specific distribution of execute threads to optimize for a specific workload. This is particularly important when certain transactions have tight response-time limits, and more threads for those transactions can be allocated accordingly. The support of multiple queues has an advantage over a single queue mechanism for shifting long response-time transactions to less critical areas.

- The database connection pool should be set equal to the number of available execute threads so that an execute thread does not need to wait for a connection.
- Experimenting with a JDBC prepared statement cache may yield a configuration that minimizes the need for parsing statements on the database. The value should be gradually increased until performance peaks.

Tuning Key Container Parameters

Many container parameters can be tuned to deploy an application more effectively to an application server. The following parameters should be considered for most applications.

- Setting appropriate session timeouts for the interval of time after which the HTTP session expires and for the idle timeout seconds to control stateful bean life cycle helps performance by making more efficient use of memory and other application server resources.
- Setting a good value for the initial bean pool size improves the initial response time for EJBs as they are pre-allocated upon application server startup.
- Setting optimal value for bean cache size will not let the server passivate beans too often, thus increasing performance by reducing file I/O activity.
- The least restrictive but valid transaction isolation level should be applied to specific EJB methods for good performance.
- Using call by reference when applicable increases the performance of method invocation.
- Configuring JSP fragment or full-page caching with appropriate, application-dependent timeout values to reduce dynamic page generation and database access requests.

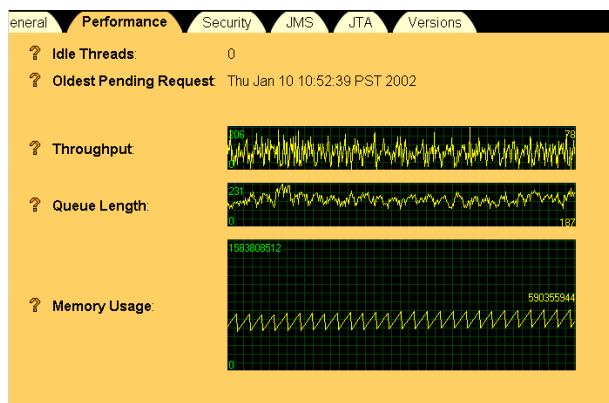


Figure 6, Case Study 2: Before application tuning. The middle chart for “Queue Length” contained values as high as 231. The queue build-up suggested that perhaps there were not enough threads to do work.

Case Study 2: Application Tuning

As an illustration of tuning application server parameters, a workload was studied using the BEA WebLogic server console. Before any tuning was applied, it was observed that while the maximum throughput was reached, the response time increased heavily with the load on the system but the processors were not fully utilized. The execute queue size was 15 while the average CPU utilization was about 70%. Using the console (see Figure 6) provided by the vendor, a high number of waiting requests was found. A considerable queue build-up was seen by the middle chart for “Queue Length.” After the disk and network I/O bottlenecks were ruled out, it was decided to increase the execute queue size and see if the performance improves. At the execute queue size of 35, average CPU utilization reached 95% and the throughput could be increased by 40%. Figure 7 shows the performance of the system through the console. The queue build-up problem was much reduced as a result of the tuning effort.

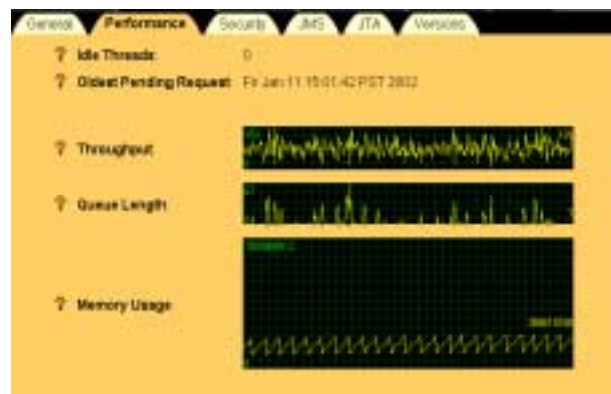


Figure 7, Case Study 2: After application tuning. The middle chart for “Queue Length” contained values mostly close to zero. The lack of queue build-up suggested that adequate threads were allocated to do work.

MACHINE-LEVEL PERFORMANCE

For applications developed using static environments such as C or C++, machine-level performance involves tuning the code for the hardware through recompilation, which complicates enterprise application deployment. With Java though, there is an additional layer – the virtual machine. This is significant for two reasons. The Java Virtual Machine (JVM) allows the application to take quick and effective advantage of new processor features since this involves only the deployment of a new JVM version and not an expensive rebuild of the entire application code.

Second, multiple versions of the application are not required to get best performance from differences in the platform such as available memory or cache. Such aspects

of the hardware are abstracted away by the JVM, and the very same version of application code can get optimal performance on the different platforms through JVM configuration tunables.

JVM-Level Performance

Selecting the correct JVM is critical. It is essential to use a JVM that has been optimized for the underlying hardware of choice. The best optimizations for various processor platforms are known [28] [29], and a Java application needs to rely on the JVM to harness these optimizations. It is also desirable that the JVM provide a rich set of configuration tunables that can be adjusted for peak performance. A JVM that can tune itself for good performance is an asset, since it simplifies deployment on a variety of platforms using the same architecture.

There are three JVM functions of interest to us: memory management, code generation, and thread management.

Memory Management

Memory management includes object allocation, heap management, and garbage collection. Modern JVMs use a variety of algorithms for these; some incorporate several algorithms for each and allow the user to select the desired one. The correct choice of algorithm is important since there are fairly significant performance differences between them. However, different techniques work better for different applications.

There are two aspects to object allocation: the management of the heap to identify the space where the object should be allocated and the preparation of the space for object allocation. The latter principally involves clearing the space by writing zeroes to it.

In a multi-processor system, the heap is shared across the processors, and obtaining space for allocation has to be a synchronized operation to ensure that no other processor is allocating an object to the same space. All synchronization is necessarily expensive, but it is especially so if the synchronized primitive is contended. A solution used by several JVMs is to allocate segments of the heap that are local to each thread, and these segments are called thread local areas (TLAs).

There are several choices for clearing space required for objects. One technique is to clear the whole TLA when it is allocated. This has the disadvantage of slowing down the object allocation that triggered the creation of a new TLA, but has the advantage that all subsequent objects created in that TLA will be able to allocate faster, since the space is already cleared. It also has the advantage of improving cache performance, since the clearing of the space serves as a prefetch from memory into the processor caches.

A second technique is to prepare the TLAs by clearing them during garbage collection (GC). It worsens the impact of GC, but all object allocation is now uniform. Finally, a third technique is to clear the space required for each object in the TLA just before allocation. This has the advantage that the object allocations are more uniform, and that neither GC nor TLA allocations take longer.

BEA JRockit, for instance, offers all three options as “cleartype:local,” “cleartype:global” and “cleartype:gc.”

The correct technique to use depends on the application. For instance, if the application puts a lot of pressure on the data bus to memory (FSB or front-side bus), then the improved cache performance of the first technique could be valuable. If uniform response time is a concern, then the third technique may be the correct choice.

There are similarly several approaches used in GC. Key aspects to consider are whether the heap should be arranged in generations and whether a significant part of GC should run concurrent with the application. It is imperative for performance that most if not all of the GC be multi-threaded. When the heap is arranged in generations, most of the GC cycles collect garbage only in the smaller nurseries, and this method is therefore not as intrusive to application performance. However, garbage is collected more frequently. Generations are effective when many objects die young, because then the GC in the small nurseries is particularly efficient. On the other hand, when GC is run concurrent with the application, the effect of each GC cycle is reduced, at the cost of a more frequent gradual impact.

Code Generation

There are two main approaches to code generation: interpreting and compiling (with a Just-in-Time (JIT) compiler). An interpreter translates each new bytecode to machine code just before execution; a compiler translates a whole segment of code (the whole application, a class, a set of classes, a set of methods, even a single method) into machine code before use.

Code compilation takes time and happens during application runtime, and the time taken to generate the code can have an impact on performance. However, the quality of code produced by the JIT is significantly superior to interpreted code, and the performance benefits of the better code should far outweigh the negative effect of compile time [30].

However, the time spent in the JIT is still an issue, and the JIT cannot therefore include all of the optimizations that a C/C++ compiler could include. This results in code that is inferior to what could have been produced if compile time was not an issue.

The solution to this is for the JVM to incorporate levels of optimizations in the JIT [31]. In other words, some portions of the application are compiled to very high quality code, whereas the remaining portions of the application are compiled quickly to lower performing code. If the portions of the application that are most used are compiled well, this will result in the whole application performing almost as well as if the whole application had been compiled thus, and it will not suffer the performance loss due to long compile times.

Many enterprise-class Java applications tend to have a large number of small methods. SPECjAppServer2002 [32], for example, has over 10,000 methods. Many compiler optimizations have a more significant impact on performance if they can operate on a larger block of code, which is impeded by the small methods. A good JIT should possess an excellent inliner to overcome this. Inlining of code during compilation is made more difficult by the large number of virtual calls in enterprise Java applications. The JIT must include approaches to de-virtualizing these calls [33].

Thread Management

A JVM either uses the threading package provided by the operating system (native threads) or it can use its own threading package and map several threads onto each kernel thread (thin threads). If the application suffers a lot from context switches, then the cost of that can be reduced by using thin threads. Similarly, if there is a pool of threads that operate on the same data, then cache performance can be enhanced by tying all the threads onto a single kernel thread. This will result in all of these threads tending to run on the same processor and benefiting from the shared data.

How a JVM handles synchronization plays an extremely important role in performance. The best way to handle a lock is to avoid locks all together, and it is good for the application developer to avoid unnecessary synchronized methods and blocks of code. In the event that such unnecessary synchronized code does exist, JVMs can possess techniques to detect them and eliminate the locks.

JVMs handle contended and uncontended locks differently, optimizing such that the uncontended lock operations go faster. The choices the JVM makes as to how it handles uncontended (thin) and contended (fat) locks, when it will promote a lock from thin to fat and when it will deflate a fat lock to a thin, and whether it will spin on a contended lock or switch over to another task can all impact performance [34].

JVM Configuration

Due to the range of choices that can be made by the JVM, a JVM can provide configuration parameters to the users

to let them identify which techniques the JVM should use for optimal performance of their application.

The more important of these parameters are all in the area of heap management, ranging from the selection of the GC algorithm and the specification of heap sizes, to the specifics of TLA sizes and when the space for an object is cleared. It is usually preferable to set the minimum and maximum heap sizes to be the same. The selected heap size can have a profound effect on performance.

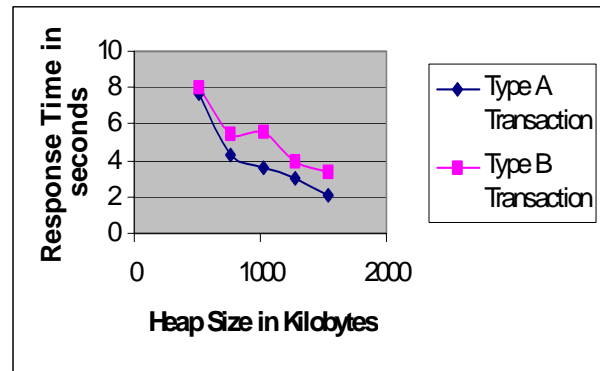


Figure 8: A variation in transaction response time for two types of transactions by changing the maximum virtual machine heap size in a typical application

The code generation can also be accessed through parameters that can decide the initial code quality and how frequently the JVM checks to see whether better code generation is warranted for a section of code, and so on.

While rules-of-thumb can be created and experience can be a guide, there is no real substitute for running a variety of experiments to identify the JVM parameters that work best for a given application.

An analogy can be made between the transmission systems in a car and JVM performance. In most cases, an automatic transmission performs adequately. For high-performance requirements such as auto-racing, however, a manual transmission works better, since, in the hands of a good driver, better performance can be had from a manual transmission. Similarly, while a good JVM will provide excellent performance as it is, appropriate selection of configuration parameters can result in even better performance.

Case Study 3—JVM Tuning

As stated earlier, heap size configuration can have a dramatic performance impact. Figure 8 shows the variation in transaction response time for two types of transactions in a typical enterprise Java application. By increasing the heap size in this experiment, we observed a two to four times improvement in response time, depending on the type of transaction.

Hardware-Level Performance

There are several hardware aspects that affect performance, including processor frequency, cache sizes, Front Side Bus (FSB) capacity, and memory speed [35]. In general we would like to get the best-possible performance on a single processor, then scale that performance across multiple processors in the box, and if more performance is desired, scale the performance out of the box by using clustering.

It is important to ensure that the settings for the BIOS and the populating of the memory sub-system follow prescribed norms. For example, a platform with 4 gigabytes of memory may perform better with four 1-gigabyte memory cards rather than with one 4-gigabyte memory card. Reading and following the system documentation can pay dividends.

Processor performance is affected most by the processor stalling, waiting on memory. The memory subsystem comprising the FSB, the server chipset and the memory cards, is typically an order of magnitude slower than the processors, and so the penalty of accessing memory for data and instruction is felt rather severely by the processors. Keeping more of the code and data near the processor, by using large caches, can alleviate this problem significantly [36].

As an experiment, we measured the performance of a typical enterprise Java application on a two-processor Intel® Xeon™ MP 2.0 GHz system with a 2 MB level-3 cache as well as on a two-processor Intel Xeon DP 2.2 GHz system without a level-3 cache. Both systems had the same amount of memory, the same operating system and application software, the same I/O connectivity, and they both had processors that included a 512 KB level-2 cache. The main difference between the systems was that one of them included an additional level of caching, a 2 MB level-3 cache. We found that having the large level-3 cache almost doubled the performance of the application.

Scaling to multiple processors in a box can be hurt by resource sharing. When the resource is a hardware resource such as the bus, larger caches can help. If larger caches are not available, or do not help, investigation into whether there is a significant impact on the bus due to data shared between threads is called for. If there is, then processor affinity could help.

If processor scaling is hurt due to software issues, it is typically a problem related to synchronization. All efforts must be made to identify the lock or locks that are the

bottleneck, and to remove them or at least reduce their use.

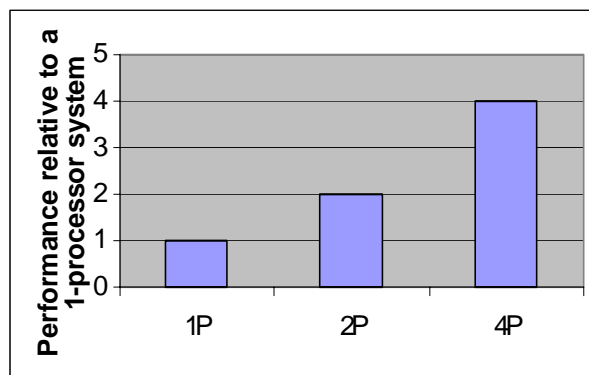


Figure 9: Performance scaling with the number of processors in a 1 GHz Itanium 2 system: performance of a 4-processor system is four times higher than a 1-processor system

One way to identify whether the processor scaling is affected by hardware or software is to measure the bus utilization. If it is high, then it most likely is a hardware resource bottleneck. Another experiment that is useful is to run the system at two different frequencies. The processor speed is now changed but not the bus or memory speed. However, the time taken to handle synchronization does scale with frequency. If now the performance scales with frequency, then it would point to a synchronization issue.

The Itanium® processor family can display excellent processor scaling. Figure 9 shows the performance scaling with the number of processors for a 1 GHz Itanium 2 system when running a typical enterprise Java application. The Itanium systems have large caches, a large capacity front-side bus, and out-of-order memory transfers, all of which enable high levels of processor scaling.

Clustering is an excellent way to increase performance when transactions share very little, frequently changed data. If there are substantial amounts of shared modified data, keeping the data coherent across the different components of the cluster will be a significant endeavor and have a big impact on performance.

It may also be possible to increase the performance within the box through clustering, by deploying more than one version of the code using multiple copies of the JVM. This has the advantage that each version of the application will run on its own copy of the JVM, with its own heap.

Intel® Xeon™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

® Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

In some platforms there is a limit placed by the operating system on heap size, and some applications do benefit from larger heap sizes. By clustering within a box, this performance benefit can be tapped.

CONCLUSION

This paper describes a top-down, data-driven, and closed-loop approach to boost enterprise Java performance. The opportunities to improve performance were examined from the perspective of the whole system, including the software/hardware stack at the system level, the software applications, and at the machine level for both the virtual machine and the physical hardware. The case studies presented in this paper suggested that all layers—not just one or two—of the system stack should be examined for performance bottleneck identification and removal.

Workloads evolved over time and no single solution works for all applications. While many good practices for enterprise Java performance were described in this paper, it is important to recognize that the performance characteristics of enterprise Java applications will change over time. The data-driven approach described in this paper should be able to adapt to the expected workload changes over time.

In the future, application servers and virtual machines may work hand-in-hand with the underlying hardware and self-tune for performance by using techniques such as dynamic feedback optimization and perhaps deriving data from some hardware performance counters. When the time comes, the approach described here may be suitable for integration into that infrastructure. Until then, our approach will help to enhance the performance of your current systems.

ACKNOWLEDGMENTS

The authors thank the members of Intel's MRTE group for performance data collection and analysis.

REFERENCES

- [1] Ken Arnold, James Gosling, David Holmes, "The Java™ Programming Language Third Edition," 2000, Sun Microsystems, Inc.
- [2] Java Community Process, "Java™ 2 Platform, Enterprise Edition 1.3 Specification," <http://jcp.org/aboutJava/communityprocess/final/jsr058/>
- [3] Kevin McIsaac, "J2EE Paves the Way to Software Infrastructure," Meta Group, August 2002.

- [4] BEA Systems Inc., "BEA WebLogic Server," <http://www.bea.com/products/weblogic/server/index.shtml>
- [5] IBM Corporation, "WebSphere Application Server," <http://www.ibm.com/software/webervers/appserv/was/>
- [6] Oracle Corporation, "Oracle9i Application Server," <http://www.oracle.com/ip/dep/ias/>
- [7] JBoss Group, "JBoss," <http://www.jboss.org/>
- [8] ObjectWeb Consortium, "Java Open Source J2EE Application Server (JOnAS)," <http://www.objectweb.org/jonas/>
- [9] Jim Farley, William Crawford, David Flanagan, *Java Enterprise in a Nutshell*, April 2002, O'Reilly & Associates, Sebastopol, CA.
- [10] Richard Monson-Haefel, *Enterprise JavaBeans*, September 2001, O'Reilly & Associates, Sebastopol, CA.
- [11] Michael Girdley, Rob Woollen, Sandra L. Emerson, *J2EE Applications and BEA WebLogic Server*, August 2001, Prentice Hall PTR, Upper Saddle River, NJ.
- [12] Mercury Interactive Corporation, "LoadRunner," <http://www-heva.mercuryinteractive.com/products/loadrunner/>
- [13] RadView Software Ltd, "WebLoad," <http://www.radview.com>
- [14] Paco Gómez, et al, "The Grinder," <http://grinder.sourceforge.net/>
- [15] Apache Software Foundation, "Jmeter," <http://jakarta.apache.org/jmeter/>
- [16] Various authors, "Open System Testing Architecture (OpenSTA)," <http://www.opensta.org/>
- [17] Intel Corporation, "VTune™ Performance Analyzer," <http://www.intel.com/software/products/vtune/vtune61/>
- [18] Borland Software Corporation, "Optimizeit Suite," <http://www.borland.com/optimizeit/index.html>
- [19] Sitraka, Inc., "JProbe," <http://www.sitraka.com/software/jprobe/>
- [20] Intel Corporation, "VTune™ Enterprise Analyzer, Java Edition," http://www.intel.com/software/products/vtune/vte_java10/

- [21] Precise Software Solutions, Inc., "Indepth/J2EE," <http://www.precise.com/Products/Indepth/J2EE/>
- [22] BEA Systems, Inc., "BEA WebLogic JRockit 8.0 (Beta) for Windows and Linux User Guide," <http://e-docs.bea.com/wljrockit/docs80/index.html>
- [23] Intel Corporation, "Performance Methodology, Terminology and Concepts," http://cedar.intel.com/media/training/perf_meth/tutorial/
- [24] Raj Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*, 1990, John Wiley & Sons, Hoboken, NJ.
- [25] Deepak Alur, John Crupi, Dan Malks, "Core J2EE patterns: best practices and design strategies," 2001, Prentice Hall PTR, Upper Saddle River, NJ.
- [26] Floyd Marinescu, *EJB Design Patterns*, 2002, John Wiley & Sons, Hoboken, NJ.
- [27] Alejandro Buchmann, Samuel Kounev, "Improving Data Access of J2EE Applications by Exploiting Asynchronous Messaging and Caching Services," in proceedings of the 28th VLDB Conference, Hong Kong, China, 2002.
- [28] Intel Corporation, "Intel Itanium 2 Processor Reference Manual for Software Development and Optimization," <http://developer.intel.com/design/itanium2/manuals/>
- [29] Intel Corporation, "Intel Pentium 4 Processor Optimization Reference Manual," <http://developer.intel.com/design/pentium4/manuals/>
- [30] A. Adl-Tabatabai et al., "Fast Effective Code Generation in a Just-in-Time Java Compiler," in proceedings of the ACM SIGPLAN'98 conference on Programming Language Design and Implementation, 1998.
- [31] M. Arnold, et. al., "Adaptive Optimization in the Jalapeno JVM," *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000)*, Minnesota, October 15-19, 2000.
- [32] Standard Performance Evaluation Corporation, "SPECjAppServer2002," <http://www.spec.org/jAppServer2002/index.html>
- [33] O. Waddell and R. K. Dybvig, "Fast and Effective Procedure Inlining," in proceedings of the 1997 Static Analysis Symposium (SAS '97), Sept. 1997, pp. 35-52.

Springer-Verlag Lecture Notes in Computer Science vol. 1302.

- [34] R. Dimpsey, et. al., "Java Server Performance: A case of building efficient, scalable JVMs," *IBM Systems Journal*, vol. 39, no. 1, pp. 151-174, 2000.
- [35] David A. Patterson, John L. Hennessy, "Computer Organization and Design: The Hardware/Software interface," 1997, Morgan Kaufmann Publishers.
- [36] C.A. Hsieh, M.T. Conte, T.L. Johnson, J.C. Gyllenhaal, and W.W. Hwu, "A Study of the Cache and Branch Performance Issues with Running Java on Current Hardware Platforms," in proceedings *IEEE Comcon '97*, pp. 211-216, 1997.

AUTHORS' BIOGRAPHIES

Kingsum Chow is a Senior Performance Engineer working with the Managed Runtime Environments group within the Software and Solutions Group (SSG). Kingsum has been involved in performance modeling and optimization of middleware application server stacks, with emphasis on J2EE and Java Virtual Machines. He has published 20 technical papers and presentations. He received his Ph.D. degree in Computer Science and Engineering from the University of Washington in 1996. His e-mail is kingsum.chow@intel.com.

Ricardo Morin is a Staff Architect working with the Managed Runtime Environments group within the SSG. Ricardo leads performance optimization activities for J2EE and JVM implementations. He has extensive experience architecting, developing, and deploying enterprise information systems. Ricardo is a Sun Certified Architect for J2EE. He received his Electronic Engineering (Cum Laude) degree from Simón Bolívar University, Caracas, Venezuela in 1980. His e-mail is ricardo.a.morin@intel.com.

Kumar Shiv is a Senior Performance Architect working with the Managed Runtime Environments group within the SSG. Kumar leads the team focusing on JVM and system performance optimization for the Itanium[®] Processor Family and earlier lead the team working on the Intel[®] Xeon[™] technology. He received his Ph.D. degree in Computer Engineering from University of Missouri in 1991, and has been a performance architect on several hardware and software projects for more than a decade. His e-mail is kumar.shiv@intel.com.

Copyright © Intel Corporation 2003. This publication was downloaded from <http://developer.intel.com/>.

Legal notices at <http://www.intel.com/sites/corporate/tradmarx.htm>.

For further information visit:

developer.intel.com/technology/itj/index.htm