Java* in a 64-bit World: 64-Bit Processing with the Intel® Itanium® Processor Family and BEA WebLogic JRockit*

By Kumar Shiv, Marcus Lagergren, and Edwin Spear

Today the 64-bit architecture provided by the Intel® Itanium® Processor Family is a viable alternative for server side applications. When Itanium® 2-based platforms are implemented with a high performing server-side Java* Virtual Machine (JVM), like BEA System's WebLogic JRockit*, the Java programming language becomes the ideal development platform for large scale, server-side enterprise class applications. These applications, which typically have large data sets, derive substantial benefit from 64-bit computing by being able to use large amounts of memory. This not only helps to reduce—and in some cases, avoid—time-consuming disk-swapping, it also makes software caching much more efficient.

Itanium 2-based systems passes beyond the sequential nature of conventional processor architectures harnessing the power of Explicitly Parallel Instruction Computing (EPIC), by allowing the software to communicate explicitly with the processor whenever operations can be done in parallel. Performance is increased by reducing the number of branches and branch mispredicts, and by reducing the effects of memory-to-processor latency. Itanium 2-based servers apply EPIC technology to deliver explicit parallelism, massive resources and inherent scalability not available with conventional RISC architectures.[1]

BEA's WebLogic JRockit is the highest performing JVM, utilizing full 64-bit functionality, on the market today. Fully optimized for the Itanium 2 microarchitecture across platforms including Windows Server* 2003 Enterprise Edition, Red Hat Linux* Advanced Server for Itanium Processor 2.1 and Red Hat Linux Advanced Workstation for Itanium Processor 2.1. BEA Web Logic JRockit leverages groundbreaking code performance and adaptive optimization, along with innovative, scalable adaptive garbage collectors to ensure optimal performance on 64-bit architectures.

Why is the Itanium Processor 2 Special?

The Itanium Processor Family boosts computing power by allowing the direct addressing of larger amounts of memory. It uses 64-bits to address virtual memory, allowing each process theoretically to access more than *4 billion times* the memory accessible by IA32 processors. In addition, the current implementation of the architecture allows the addressing of an equally massive amount of physical memory. In comparison, Intel® Xeon™ processor-based systems allow four gigabytes of virtual memory per process.

Server applications tend to exploit the large amount of memory 64-bit processors provide. Since they access considerable amounts of data, the availability of a large address space common to 64-bit processors significantly reduces disk accesses. Additionally, it allows caching of network accesses, potentially reducing that as well. We

---

[1] From Intel Press Release, "HP and Intel Unveil Breakthrough EPIC Technology at Microprocessor Forum", 14 October 1997.

thus see that benchmarks such as TPC-C perform better on 64-bit architectures such as that of the Itanium 2 processor.

Java and 64-bit Computing

Java applications, particularly server-side applications, are well-suited to run in a 64-bit environment. Java applications allocate objects into a heap, and as a rule, the rate of this object allocation is high. When the heap is full, garbage collection must occur so that heap space can be freed, allowing the application to continue to run. Large Java applications benefit by having larger heaps because the often-expensive overhead required for garbage collection is reduced by reducing how often it happens. Moreover, a larger heap might allow the JVM more flexibility in finding a less intrusive point at which to collect garbage.

Success of 64-bit computing relies on how well the system addresses a number of challenges. With Itanium 2 processors, the compiler has perhaps an even greater responsibility than on other platforms, because the EPIC concept allows the compiler significantly more room to extract benefit from the architecture.

Computer performance has always depended on both the quality of the code generated and the ability of the underlying hardware to quickly consume the code. The Itanium 2 processor's EPIC architecture bundles multiple instructions together and then execute the bundle during the same clock cycle. The compiler identifies which instructions can be bundled together to take advantage of this parallelism.

The Importance of the JVM

An important reason for the popularity of Java is its platform independence, where applications developed on one platform can be deployed on any other platform. The applications implicitly depend on the JVM to provide the optimal performance for the platform. Code generation, thread management, memory allocation and garbage collection are all very important aspects of Java application performance, and how well a JVM handles them can be a key performance differentiator.

In Java, code generation occurs during application execution; therefore, not only must the compiler generate good code but, for Java, it must also generate that code quickly. A slow code generator will negatively impact application performance. Additionally, application start-up is also undesirable.

The compiler's code scheduler is a key component. It needs to look at available instructions and decide which of them can be bundled together. The more instructions that it can observe, the better the chances of finding code that effectively can be bundled and benefit from code parallelism. The "scope," or the amount of code, that the code scheduler has visibility into, is thus an important factor in application performance.

Java code characteristically consists of a large number of classes and methods, the majority of which tend to be very small. The sizes of the methods sharply limit the scope seen by the scheduler. The JVM needs to find ways to expand the scope.

The JVM also must be capable of efficiently handling a large number of methods. Enterprise Java applications usually do not have overly hot sections of code; for example

you will note that the SPECjAppServer2002 benchmark has about 10,000 methods, none of which is responsible for more than 3% of the execution time.[2]

The memory manager also has challenges to face: while the larger heap made possible by the 64-bit architecture might benefit performance, it also requires that the JVM have heap management algorithms that scale well, particularly on multiple CPUs. Heap fragmentation is always an issue, but even more so when the heaps are large. A larger heap allows for larger applications with fewer garbage collections, at the expense of longer garbage collection times. With 64-bit computing, the classic "stop-the-world" strategies are no longer effective, as the pause times resulting from these strategies would be undesirably lengthy.

Why Choose JRockit?

As we have seen, the keys to top performance by an application server are how application code is generated and how well the memory system is managed. Of all the JVMs on the market today, only BEA WebLogic JRockit provides the kind of performance necessary to meet the rigorous challenges of 64-bit, server-side computing. BEA WebLogic has been designed for server-side use since its inception in 1998, providing code generation prowess and intelligent optimization techniques that take full advantage of the Itanium 2 processor's vast technological advances. Additionally, its innovative scalable adaptive garbage collectors make memory management a thorough and efficient process that results in minimal impact during program execution.

Compiling 64-bit Applications with JRockit

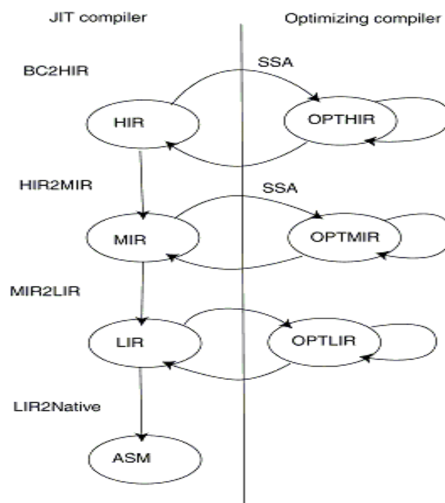
In the interest of quick start-up, many JVMs choose first to interpret the Java byte code and then compile them later during the run. BEA WebLogic JRockit, however, compiles the code at first use by implementing a just in time (JIT) compiler. This ensures desirable application performance from the outset, albeit at the cost of a slightly longer start-up time. To expedite start-up, however, BEA WebLogic JRockit does not use all possible compiler optimizations. While doing so might lead to even better performance early in the application run, it results in a slower start-up, which is certainly undesirable.

Compiling all of the methods with all available optimizations is also undesirable. Since the compilation time is part of application execution time, compiling all of the methods with all available optimizations also negatively impacts application performance. To address this issue, BEA WebLogic JRockit does not fully optimize all methods at start-up; in fact, it leaves many methods unoptimized throughout the entire application run. Instead, BEA WebLogic JRockit chooses those functions whose optimization will most benefit application performance and only optimizes those few methods.

BEA WebLogic JRockit can thus be seen to have two distinct, but cooperating, code generators, a JIT compiler, which resolves data from bytecode, through three levels of intermediate representation, to native code (assembly language), and an optimizing compiler, which optimizes *targeted* methods at each level of intermediate representation.

---

[2] "A Performance Study of SPECjAppServer2002", Kumar Shiv, Intel Corp., First Workshop on Managed Run Time Environment Workloads at CGO 2003, March 2003.



BEA WebLogic JRockit Code Optimization Paths -- Most methods will only traverse the left half of the diagram while methods targeted as "hot" will also spend time in the optimizing compiler.

BEA WebLogic JRockit uses a sophisticated, low-cost, sampling-based technique to identify which functions merit optimization: a "sampler thread" wakes up at periodic intervals and checks the status of several application threads. It identifies what each thread is executing and notes some of the execution history. This information is tracked for all the methods and when it is perceived that a method is experiencing heavy use—in other words, is "hot"—that method is earmarked for optimization. Usually, a flurry of such optimization opportunities occur in the application's early run stages, with the rate slowing down as execution continues.

Since method sizes tend to be small, and scope is so important to the code scheduler, method inlining is the single most important optimization tool available. "Inlining" simply means that the code of a called method is inserted directly into the call site. This can be difficult to do in Java for a number of reasons; for example, interface calls, remote calls, and virtual calls might call functions the identities of which are not known until execution time. Although inlining can be a very effective optimization technique, when poorly implemented, it can lead to code-bloat and to a drop-off in performance. BEA WebLogic includes well-tuned heuristics that prevent such performance loss.

The optimizing compiler in BEA WebLogic JRockit includes many of the best-known techniques for code generation on Itanium 2 systems. This includes a sophisticated register allocator that takes full advantage of the Itanium 2 processor's large register stacks (128 general purpose and 128 floating point).

Memory Allocation and Garbage Collection

The heap management policies of BEA WebLogic JRockit scale very well with the number of threads and with 64-bit's larger heap size. Memory allocation is done using "thread local arrays," where each thread is assigned a small heap into which it allocates

objects. With thread local arrays, space for about a thousand objects is allocated on the heap on behalf of each thread. This scheme promotes spatial and temporal locality of the data, leading to excellent performance of the processor caches. It also sharply reduces the synchronization between threads to acquire the heap to allocate objects. The size of the thread local array is an important parameter for performance, and the optimal size is application dependent. BEA WebLogic JRockit includes heuristics that tune this parameter at execution time.

BEA WebLogic JRockit recognizes that applications have different needs and thus, a garbage collector that works well for one application might work very poorly for another. To provide high performance across a wide variety of applications, BEA WebLogic JRockit supports four different garbage collectors:

- **Generational Copying** divides the memory into two or more areas called "generations". Instead of allocating objects in one single space and garbage collecting that whole space when it gets full, most of the objects are allocated in the "young generation", called the nursery.
- **Single Spaced Concurrent** removes garbage collection pauses completely. This allows for gigabyte-size heaps and no long pauses. However, concurrent garbage collectors trade memory throughput for reduced pause time.
- **Generational Concurrent** allocates objects in the young generation. When the young generation (the nursery) is full, the JVM "stops-the-world" and moves the objects that are still live in the young generation to the old generation. An old collector thread runs in the background all the time; it marks objects in the old space as live and removes the dead objects, returning them to BEA WebLogic JRockit JVM as free space.
- **Parallel** stops all Java threads and uses every CPU to perform a complete garbage collection of the entire heap. A parallel collector can have longer pause times than concurrent collectors, but it maximizes throughput.

The JVM includes heuristics to adaptively find the optimal garbage collection algorithm for each application. All of the garbage collectors have been designed to handle large heaps well, and the algorithms take advantage of the sparseness of the heap; that is, large amounts of the heap are garbage because most of objects are short-lived.

The garbage collectors are differentiated based these characteristics:

- Do they include a nursery (generational or not)?
- Is the marking phase multi-threaded?
- Is the sweeping phase multi-threaded?
- Does the collector run concurrently with the application or does it stop the application during garbage collection?

These characteristics can affect the frequency and duration of garbage collection. From an application throughput standpoint, the appropriate garbage collector should be that which minimizes the total garbage collection time; that is, the frequency of garbage collection multiplied by the average duration of each collection. However, in many

applications, the response time is important as well. In those cases you want to choose a garbage collector that will minimize pause times. BEA WebLogic JRockit allows the user to indicate whether response time is important for the application or whether throughput is the only consideration.

With the large heaps available to 64-bit systems, fragmentation can become a serious performance issue. Compacting the heap during garbage collection alleviates this problem, but will impact performance since compacting large heaps is expensive. Avoiding compaction altogether is problematic in that, unless compacted, portions of the heap will become unusable, leading to more frequent garbage collection. Locality, too, can impact the performance of the processor caches. BEA WebLogic JRockit solves compaction problems by using a sliding compaction window. During each garbage collection a small part of the heap is compacted—a different part being compacted each collection. With a properly sized window, the heap performs as well as it would with full compaction, while the cost of garbage collection stays as small as it would with no compaction.

Into the Future with JRockit

BEA WebLogic JRockit includes many of the most useful compiler techniques developed for the Itanium processor. It takes effective advantage of the register stack and the multiple execution units. Through a sampling technique, it can profile the application and use this information to identify which methods to optimize. It includes two levels of code scheduling to ensure optimal bundling of instructions. It offers the user a variety of garbage collectors, as well as the ability to adaptively pick the best one for the application. BEA WebLogic JRockit scales very well with the number of processors, not imposing any unnecessary synchronization burdens on the system. It takes full advantage of the large-capacity, out-of-order front-side bus to memory, and a scaling of nearly four has been achieved in some workloads when growing from one processor to four.

Looking into the future, many enhancements are planned for BEA WebLogic JRockit. While the information provided by the hot-spot sampling technique is already popular, even more information can be extracted to provide guidance to the optimizing compiler. The resulting optimizations should lead to even better inlining and an increase in the bundling efficiency of the scheduler. Additionally, BEA WebLogic JRockit is experimenting with a unified garbage collector that will unite the parallel and concurrent approaches with memory management. BEA is also involved in research to combine this unified collector with a reinforcement learning approach to ensure superior runtime feedback.

With the Itanium 2 Processor and JRockit, Java *is* the Language of the 64-bit World

As we've demonstrated in this report, the 64-bit world presents innumerable opportunities for Java to become the de facto programming language for large-scale, enterprise-level applications. It also presents a host of challenges that must be surmounted to make this a reality. Fortunately, the combination of the Intel Itanium Processor Family's EPIC architecture and the powerful engine of the BEA WebLogic

JRockit JVM have been well designed with the necessary processing capabilities and innovative code and memory handling to meet those challenges.

About the Authors

Kumar Shiv is an Intel Corporation Senior Staff Performance Architect working with the Managed Runtime Environments group within the Software and Solutions Group. Kumar leads the team focusing on JVM and system performance optimization for the Itanium® Processor Family and earlier led the team working on Java performance on the Intel® Xeon™ processor. He received his Ph.D. degree in Computer Engineering from the University of Missouri in 1991, and has been the lead performance architect on several hardware and software projects for more than a decade.

Marcus Lagergren is BEA Systems' Senior Software Engineer working on the BEA Weblogic JRockit JVM. He is one of the principal architects of the BEA JRockit code generator and has been working on the JRockit project since January 1999. Marcus holds an M.Sc in Computer Science from the Royal Institute of Technology in Stockholm.

Edwin Spear is a BEA Systems' senior staff writer working on the BEA WebLogic JRockit SDK. Since 2001, Edwin has authored Java application and development documents for numerous BEA products, including WebLogic Application Integration and WebLogic Portal. He holds a Bachelors degree in English-Written Communications from Santa Clara University.

Information in this document is provided in connection with Intel® products. Except as provided in Intel's terms and conditions of sale for such products, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY RELATING TO SALE AND/OR USE OF INTEL PRODUCTS, INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life-saving, life-sustaining, critical control or safety systems, or in nuclear facility applications. Intel may make changes to specifications, product descriptions, and plans at any time, without notice.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/procs/perf/limits.htm, or call (U.S.) 1-800-628-8686 or 1-916-356-3104.

Intel, Itanium, and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.