USENIX Association

Proceedings of the Third Virtual Machine Research and Technology Symposium

San Jose, CA, USA
May 6–7, 2004

Java™ Just-In-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications

Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark Stoodley, and Vijay Sundaresan
Toronto Lab, IBM Canada Ltd.

Abstract

This paper describes optimization techniques recently applied to the Just-In-Time compilers that are part of the IBM® Developer Kit for Java™ and the J9 Java virtual machine specification. It focusses primarily on those optimizations that improved server and middleware performance. Large server and middleware applications written in the Java programming language present a variety of performance challenges to virtual machines (VMs) and just-in-time (JIT) compilers; we must address not only steady-state performance but also start-up time. In this paper, we describe 12 optimizations that have been implemented in IBM products because they improve the performance and scalability of these types of applications. These optimizations reduce, for example, the overhead of synchronization, object allocation, and some commonly used Java class library calls. We also describe techniques to address server start-up time, such as recompilation strategies. The experimental results show that the optimizations we discuss in this paper improve the performance of applications such as SPECjbb2000 and SPECjAppServer2002 by as much as 10-15%.

1. Introduction

The steady growth in the size and complexity of large server and middleware applications written in the Java™ programming language[8] offers continuing performance challenges to virtual machines (VMs) and just-in-time (JIT) compilers. One source for these challenges is that program maintenance has become at least as important as performance for this class of applications. This stronger emphasis on program maintenance has encouraged programmers to make more widespread use of the many features available in the Java programming language that support software engineering efforts, such as: multiple inheritance via interface classes, polymorphic virtual method invocations, and complex exception handling mechanisms such as `finally` blocks. While these features are convenient to use from a Java programmer's perspective, they can impose a significant runtime performance overhead and thus they challenge JITs and VMs to provide features that mitigate that overhead. Moreover, addressing these software engineering related overheads is doubly difficult since many large middleware applications do not have obvious hotspots on which to focus compilation or performance analysis resources.

Even beyond this particular challenge, however, server and middleware applications have many other features that negatively impact their performance. In this paper, we describe 12 separate optimizations and features that have been implemented in IBM products to accelerate the performance of these applications. Examples of these features include: optimizing synchronization to reduce repetitive locking and unlocking on the same object to improve scalability, and optimizing Java class libraries to improve the performance of, for example, computing transaction time-stamps.

Steady-state performance, however, is not the only important performance factor. Application servers, for example, must be able to start quickly when a machine is rebooted to avoid costly interruption of service. Fast start-up time also greatly enhances productivity for application developers who regularly must restart the application server as part of the usual edit-compile-debug development cycle. We show in this paper how the JIT compiler can employ different recompilation strategies to significantly improve server start-up time.

This paper describes recently implemented optimization techniques used by the Just-In-Time compilers that are part of the IBM® Developer Kit for Java and the J9 Java virtual machine specification, focussing primarily on those optimizations that improved server and middleware performance. We report performance results on three

platforms for industrial-strength implementations of these features rigorous enough to stand up to our customers' applications.

The rest of the paper is organized as follows. In Section 2, we discuss three Java coding conventions we have observed in customer applications and detail some of the performance overheads incurred in each case. In Section 3, we describe eight optimizations that specifically target server performance. Similarly, in Section 4, we describe four optimizations we have found to be important for middleware applications. We present the performance results achieved by these techniques in Section 5 using a well-known server application (SPECjbb2000), a middleware application (SPECjAppServer2002), as well as an XML parser and a cryptographic benchmark program. Finally, in Section 6, we summarize the paper's contributions.

2. Java Coding Observations

Through the process of addressing customer defects, our teams are exposed to large amounts of Java code written by our customers. We present, in this section, what we believe are three important observations about the code being employed that have a direct impact on the performance of Java applications: 1) a trend towards bytecode generation rather than Java code translated by `javac`, 2) the more frequent use of `finally` blocks, and 3) the common use of exceptions.

2.1. Bytecode Generation

Bytecode generators other than `javac` are becoming more prevalent. These range from relatively simple tools such as JavaServer Pages (JSP) servlet generators to full-fledged compilers such as extensible stylesheet language transformation (XSLT) compilers. The bytecode streams generated by these tools may have performance implications throughout the virtual machine and in particular in their interactions with JITs. Some of these interactions can be ameliorated through advances in JIT implementations but some require care in the bytecode generator implementation.

One particular factor that should be taken into account is that very long compilations can cause undesirable system behavior. For example, if a very large method is invoked as part of servicing an RMI request, the time taken to compile that large method might trigger a time out elsewhere, the effect of which could cascade through the system. For this and other reasons, most JITs have built-in limits on various phases such as method inlining to prevent compile times from becoming prohibitively long. However, methods that are individually very large, such as some generated JSP servlets, can still induce long compilations, which can result in performance problems as outlined above.

A second important consideration is that automatically generated bytecodes tend to exploit the more powerful aspects of the Java programming language more fully than programs written directly in Java. Tools that compile domain-specific languages to bytecode quite naturally build their own abstractions to represent elements of interest in their language. For example, if language X supported subroutine calls, an X-to-bytecode compiler might use an explicit representation of a stack frame. It is often easier for such a compiler to use type-opaque representations of these abstractions, which are supported by the Java language, to simplify both the bytecode generation model and the supporting run time. JITs, on the other hand, are better able to optimize the use of concrete data types and abstract types typically used by Java programmers. Work may be required in both JITs and bytecode generators to ensure that generated bytecodes benefit from overall optimizations such as de-virtualization and method specialization.

2.2. Finally Blocks

`Finally` blocks provide a mechanism whereby a Java programmer can guarantee an action is performed regardless of how control leaves a method: whether via a normal return path or via an exception. One way that this mechanism is employed in middleware applications is to guarantee that a tracing function will be called if it is necessary to document that the method has completed executing. In this coding pattern, most of the code in the method is enclosed within a `try` region and the tracing code is placed in a `finally` block for that `try` region. This organization guarantees consistency of tracing information.

The control flows associated with the use of a `finally` block can be quite complex, complicating the construction of efficient and effective traversal orders for the JIT, which can necessitate more iterations than otherwise necessary to solve dataflow equations. The resulting increase in JIT compilation time can affect the program's performance. Even worse, if the control flows are sufficiently complex (consider nested `finally` blocks, for example, which can and do occur after aggressive inlining) the JIT may decide not to, or be unable to, apply certain optimizations that can have profound performance implications.

Although `finally` blocks are convenient from a programmer's point of view, they are less so for the JIT compiler. We recommend against using `finally` blocks in performance-critical codes unless there is a valid engineering reason to do so.

2.3. Exceptions

Despite the wealth of evidence detailing the hidden cost of processing exceptions, we still encounter applications where exceptions are thrown in commonly executed paths. While JITs expend considerable effort to improve the delay between the time an exception is thrown and the time that the handler for that exception begins to execute, the delay is still significant and several orders of magnitude larger than executing, for example, a branch. In particular, throwing an exception that will be caught higher in the stack can be particularly expensive because each frame must be searched both for a catcher and for locked objects that must be unlocked. The latter problem is typically addressed by having an artificial catch block that unlocks the object and then rethrows the exception. If the "real" catcher is many frames above the frame where the exception was originally thrown and there are many objects to unlock in the intermediate frames, the exception will be even more expensive to process; it will involve throwing the exception several times.

3. Server Performance Work

In this section, we describe eight areas in which we have made enhancements (either in the VM or in the JIT) to improve the performance of server applications: `newInstance`, `String.indexOf()`, `System.currentTimeMillis()`, `System.arraycopy()`, object allocation inlining, lock coarsening, thread-local heap batch clearing, and utilizing the Intel® SSE instructions. Many of these enhancements target, in particular, the SPECjbb2000 [16] benchmark.

3.1. newInstance

The `java.lang.Class.newInstance` method returns an instance of the same type as the `java.lang.Class` object passed as the parameter to `newInstance`. The `newInstance` method in the class library calls a native `newInstanceImpl` method that does three things: first, it checks for the presence of a default constructor for the class being instantiated; second, it checks that this default constructor is accessible to the caller; and third, it invokes the constructor. Several factors make this process inefficient:

1. The expensive invocation to the native `newInstanceImpl` method.
2. The search for the default constructor.
3. Verifying that the constructor is visible in the caller's context may involve walking the stack to determine that the constructor is not currently being executed.
4. The expensive callback into the Java constructor from the native method.

Because `newInstance` is commonly used, it warrants a special slot in the virtual function table (VFT) of each class. This entry represents a class-specific `newInstanceImplThunk`, which can perform the allocation, avoiding both the native invocation (#1) and the callback to the Java constructor (#4). We also avoid the accessibility check (#2) altogether for classes with public default constructors. Furthermore, we can inline the constructor into the caller and avoid even overhead of a JITed code invocation. Even better, the JIT can use value profiling information[5] to inline the `newInstanceImplThunk` invocation with appropriate runtime guards, which has the benefit of exposing the allocation to the JIT's optimizer. Once exposed, optimizations such as escape analysis[6] can transform the heap allocation into a stack allocation if the created object does not escape to another thread or to the method's caller.

3.2. String.indexOf()

Numerous algorithms, such as Boyer and Moore's technique[4], exist to perform the semantics specified by the `String.indexOf()` method in the Java class library. Many of these algorithms operate in two phases: first they compute some meta-data based on the target string and then they use this to rapidly iterate through the source string. Our analysis of benchmarks such as SPECjbb2000 revealed that `indexOf` is frequently invoked with short constant pattern strings. This situation is ideal for the application of Boyer and Moore's algorithm because the compiler can statically compute the meta-data associated with the pattern string, leaving only the rapid phase to be executed in response to the actual invocation of `indexOf`.

3.3. System.currentTimeMillis()

The `java.lang.System.currentTimeMillis()` method is often called by transaction-based server applications such as SPECjbb2000 that repeatedly build time stamps. This method is expensive because:

1. There is an expensive call from JITted code to a native method.
2. The method returns a long value that must be held in a register (increasing register pressure) or returned via the stack (which may be slower to access).

We optimize these costs by generating a platform-specific inline code sequence to replace calls to `System.currentTimeMillis()`. The inline sequence sets up the arguments according to the system linkage convention and then directly calls the appropriate OS timing routine: `GetSystemTimeAsFileTime()` in a Microsoft® Windows® environment, for example, or `gettimeofday()` in a Linux environment. The inlined sequence stores the result directly as required by the application's use of `currentTimeMillis()`.

3.4. System.arraycopy()

One of the most frequently used intrinsics in Java middleware applications is `System.arraycopy()`. The optimal code sequence for this method is platform-specific and varies with the actual size of the array to be copied.

To avoid the frequently suboptimal generic arraycopy implementation for specific calls to `System.arraycopy()`, the compiler can either generate code inline or invoke tuned versions of `System.arraycopy()` provided in the compiler's runtime. The compiler can, of course, implement both approaches and evaluate the cost trade-off at code generation between the call overhead to the tuned library code versus the code expansion cost of the inline code to copy the array.

If the size of the array to be copied is constant, the JIT can easily generate the best possible instruction sequence. When the size isn't known, however, it has to either generate an instruction sequence that will perform best on average (like the generic implementation of the arraycopy intrinsic) or it can employ value profiling to determine the most frequent array size. In a subsequent compilation for the method, the JIT can then special-case that most frequent size with an inlined code sequence and rely on a more generic code sequence for other sizes. Since most of the arrays copied in middleware applications are shorter than 256 bytes, the JIT can also choose either to use a general instruction sequence that works best for shorter array sizes, or to use a runtime test to exploit the benefit of the faster arraycopy instruction sequence for cases where the faster sequence will compensate for the test overhead.

When copying arrays of references, there are additional optimizations that are performed. These optimizations fall into two broad categories deriving from: 1) the garbage collector design, and 2) the need to enforce the Java rules [11][12] for reference assignment compatibility.

To maintain correctness, a generational garbage collector[18] must detect when a reference to a nursery (or "new space") object is stored into a tenured (or "old space") array object. For certain array copies, the JIT can prove that checking for such stores is unnecessary. The simplest optimization opportunity of this class, for example, is copying regions within a single array. Clearly the array cannot be both in the nursery and tenured at the same time. A second opportunity in this class occurs once one nursery object reference has been stored into a tenured object/array. Once the tenured array has been added to the list of objects that must be scanned when garbage collecting the nursery, the remaining elements of the array are copied without checking for nursery objects[1].

The second class of optimizations results from the JIT's existing type propagation capabilities. It is often the case that the JIT is able to prove that there will be no exceptions raised by storing the elements of one array into another.

There are two effects of these optimizations that improve performance. First, proving that it is unnecessary to load either the class object or the garbage collection bits from the header of the object being copied reduces memory traffic into the cache during the copy, which in turn reduces the cache pollution effects of the copy. Second, by eliminating the control flow from the copy loop, more efficient copy mechanisms supported by most CPU architectures are enabled.

3.5. Object Allocation Inlining

Virtually all versions of the IBM Developer Kits have included facilities that enable JITs to perform common object allocations using a thread local heap (TLH). A TLH is a small region of the heap that is temporarily assigned for the exclusive use of a particular thread. When no further allocations are possible in a TLH, the region resumes its status as a regular part of the heap.

---

[1] The check can be omitted so long as collection cannot occur during the array copy.

Since each thread has its own TLH, no synchronization is needed to allocate the majority of objects. When processing modestly sized object allocation requests, the JIT will generate fast-path code to allocate the objects in the TLH. The initial implementations of this strategy targeted the case where the allocation is satisfied from the TLH as the fast-path. The generated code was quite effective in this case, but each allocation site had associated with it a series of return code checks and recovery sequences to handle the less-frequent cases where the allocation could not be satisfied by the TLH.

Analysis of many benchmarks revealed that the execution time spent in the return-code checking code could be substantial. The Developer Kit was enhanced so that, when the fast-path code fails to allocate storage from the TLH, it invokes a JIT service routine. This routine, which is part of the garbage collection (GC) component, will always return either a new object or an `OutOfMemoryError` indication. The latter can be checked quickly by the fast-path code. The service routine handles the necessary bookkeeping, including possibly collecting unused objects, as well as memory synchronizations. As a result, the fast-path code sequence is shorter, which results in better instruction cache utilization. Furthermore, in most cases where the original TLH storage allocation request fails, fewer compares and branches are executed.

3.6. Lock Coarsening

Because the JIT aggressively inlines invocations when compiling a method, it is not uncommon to synthesize a block of code containing several lock and unlock operations applied serially (i.e., not in nested fashion) to the same object. These lock and unlock operations result from inlining synchronized methods into the method being compiled. When we analyzed SPECjbb2000, we noted a method that executed 6 or 7 repetitive lock and unlock operations on the same object, depending on the path executed.

Synchronization has two negative impacts on program performance. First, the lock and unlock operations themselves are expensive, in most cases requiring expensive atomic memory accesses. Second, locks act as barriers to optimizations in the JIT, to processor instruction schedulers, and to the reordering ability of most modern processors. To alleviate these two problems, we implemented a lock coarsening optimization that eliminates many of the intermediate unlock and lock operations, leaving only one lock to be executed early in the method and as many unlocks as there are paths exiting the locked region. This optimization must be performed with great care, however, because holding a lock for a substantially longer time can increase contention and therefore reduce an application's scalability. Even worse, without proper care, deadlock opportunities can be created that make the application hang.

We implemented a computationally efficient lock coarsening pass that does not unduly increase contention or create deadlock opportunities. The pass only acts on synchronization resulting from inlining synchronized methods and so it will not interfere with hand-crafted synchronization implemented by a programmer via a synchronized block. Although there has been previous work in this area [1][3][7], our implementation is novel because of its aggressiveness and its computational efficiency. Unfortunately, space restrictions prevent us from providing a detailed description of the technique in this paper.

3.7. Thread-local Heap Batch Clearing

The IBM JITs employ a thread-local heap (see Section 3.5) to reduce contention on the heap lock. On some processors, it can be more efficient to initialize the entire contents of the thread-local heap to the value zero when it is allocated rather than initialize those values for each individual object allocation. In particular, the IBM PowerPC® architecture allows an entire cache line at a time to be initialized to zero. The IBM JITs use this batch clearing approach only for processors that have efficient architectural support for initializing large blocks of memory. For other processors, the objects allocated from the local heap are initialized individually.

3.8. Intel® SSE Instructions

When Intel introduced the Pentium® III processor, it included Streaming SIMD Extensions (SSE), a technology designed to accelerate applications such as 2D/3D graphics, image processing, and speech recognition. It includes a file of eight 128-bit XMM registers. The SSE architecture specifies that each XMM register holds four single-precision floating-point values. A further extension (SSE2) introduced in the Xeon™ and Pentium 4 processors, allows each register to hold a pair of double-precision floating-point values (or four single-precision values). The extensions include a rich set of instructions for inserting and extracting data from XMM registers and, of course, for manipulating the data therein.

Fully exploiting the SIMD capabilities of these extensions in a Java environment is nontrivial. The difficulty arises primarily because the extensions

perform best when the data being moved in and out of XMM registers have specific alignment characteristics and the instructions are used in a streaming fashion. Keeping salient data aligned in an environment with active garbage collection (and object relocation) is a significant challenge. Furthermore, relatively few applications are streaming in nature. Without surmounting this challenge, however, the SSE architecture can still be used to good advantage. If each XMM register is treated as a single value (single- or double-precision), the extensions form an excellent scalar floating-point computation engine. Furthermore, it is easier for the JIT to manage a set of eight orthogonal registers than the x87 FPU stack, especially in the presence of registers whose live ranges span control flow boundaries. Our JIT compiler therefore favors the SSE extensions over the traditional floating point mechanisms when targeting X86 CPUs. An exception to this rule is in cases where SSE2 is not available and a particular method requires frequent conversions between single and double precision.

4. Middleware Performance Work

We describe in this section four items on which we have recently focussed to improve the performance of middleware applications like SPECjAppServer2002[15]: reducing application server start-up time via recompilation strategies, improving interface invocations via polymorphic inline caches, recognizing when 64-bit long variables are used to (inefficiently) perform unsigned 32-bit computations, and code reordering to reduce branch mispredictions and instruction cache misses.

4.1. Application Server Start-up Time

The start-up time of the application server is critical for many reasons, including quick recovery from server failures and shorter development time since the server is often started at least once per edit-compile-debug cycle.

This requirement is a challenging aspect of the quality of the JIT compiler because the speed of the compiler itself is critical to good performance, rather than the quality of the generated code. There are two conflicting factors that have significant effect on the start-up performance. First, the compiler itself should take as little time as possible. Second, we want to generate fast code to reduce the application execution time. These two factors must be carefully balanced and tuned to minimize start-up time.

We approached this problem by applying various recompilation strategies with different optimization levels and correspondingly different compile times. However, application server start-up frequently involves transient behavior: some methods are hot for a short period of time, which confuses the profiler and causes the JIT to recompile the method at an optimization level higher than is truly justified. The JIT expends considerable resources compiling such a method, but the method does not execute frequently after it is recompiled and so the compilation effort is not rewarded. To reduce incidences of this problem, the JIT compiler has additional heuristics to estimate the expected gain to be achieved by recompiling methods at higher optimization levels. These heuristics dampen the aggressiveness of compilation optimization early in the process, which benefits application server start-up time.

4.2. Polymorphic Inline Caches

Java invocations can be polymorphic: when a method is invoked, the actual target is determined by the runtime type of the receive object. This feature benefits programmers by allowing the design of clear, reusable and maintainable code. But this benefit comes with the additional and sometimes not insignificant cost that each such polymorphic invocation requires the system to decode the type of the receiver object before the appropriate target can be invoked. Determining the correct target for an invocation via an interface method can be particularly expensive because the Java language allows a class to implement multiple interfaces. Large middleware applications often use interface-based polymorphism so that they can be easily extended with additional features, and therefore these applications can suffer from the performance impairment because of the polymorphic call overhead

An efficient technique to reduce the effect of the interface invocation overhead is a polymorphic inline cache (PIC) [2][9][10][17] which is a dynamic code or data structure that can cache a particular target call site and perform quick subsequent invocation at the same call site, without invoking the target lookup routine. The compiler generates and maintains a small cache containing usually two to four entries of the most frequent actual targets for the polymorphic invocation. By employing such a cache for each interface invocation site, the majority of the invocations can be quickly dispatched. Since most polymorphic invocation sites have only a few targets[2], caching a small number of targets for low-overhead invocation can greatly mitigate the total overhead of the full polymorphic call sequence.

The polymorphic cache is initially empty after the method is compiled. As the program continues to

execute, the targets invoked from the site are recorded with the type of the invocation's receiver object into the cache. Subsequent invocations with that same receiver object type will be directly dispatched to the cached target. If the cache is full and the receiver object's type does not match any of the cache entries, a full lookup and dispatch is performed (the "slow" path). Profiling can be used to order or even evict entries in the cache according to frequency. The profile data can also be used to recompile the method containing the call site if inlining the most frequently executed target would be beneficial.

4.3. Unsigned Arithmetic for Cryptography Applications

With the introduction of SSL (Secure Sockets Layer) and its implementation in middleware application servers written in Java code, the performance of transactions that use the security protocol becomes as important as traditionally non-secure transaction operations. This secure layer overhead can significantly impair the performance of a secure application compared to a non-secure implementation.

The JIT can affect the performance of this secure layer because of a specific characteristic of the Java code often used to implement these cryptographic codes. A frequently appropriate data type to use in these codes is unsigned 32-bit integer, which is unfortunately not directly available in the Java programming language. To implement many cryptographic algorithms, therefore, Java programmers have resorted to using the 64-bit signed long data type to hold 32-bit unsigned data, which on 32-bit architectures can cause a significant performance slowdown.

To address this problem we have added support in the JIT compiler to recognize long arithmetic operations that are used to implement 32-bit unsigned arithmetic. If the JIT can prove that the upper 32 bits of a 64-bit computation are zero, then more efficient instruction sequences can be used on some platforms to accomplish that computation than the naïvely generated 64-bit code.

4.4. Code Reordering

As a debugging aid, the methods of many middleware applications include tracing code that conditionally executes at the beginning and/or end of the method to document how the application is executing. Since the tracing information is typically voluminous, this code is typically executed when debugging the application and is only rarely executed in a production environment. An example of this style of coding is as follows:

```
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
   }
}
```

While these conditional code snippets provide useful debugging information when there is a problem, they can also impose a performance penalty in a middleware application even when they do not execute. There are three aspects to this penalty:

1. The additional code can perturb the JIT's allocation of machine resources such as registers.
2. The instruction fetch engine for many modern processors will (incorrectly) predict by default that the forward branch around the code snippet will not be taken.
3. The effective user code is inefficiently and unnecessarily spread over additional cache lines and memory pages.

Fortunately, the first problem is easily overcome because the JIT usually has access to profile information indicating which paths are not frequently executed. In many cases, this information enables the JIT to avoid dedicating resources to those parts of the code that are rarely executed. In the above example, the JIT would not likely inline the `entry()` and `exit()` invocations if those paths are marked as rare code.

In contrast to the first problem, the second problem is significant. The condition test is turned into a forward branch around the code outputting the trace information. Most current processors, by default, assume that forward branches are not taken. To alleviate this problem, current processors also include prediction tables that remember what happened the last time a branch was executed. These tables are used by the processor to predict future outcomes for branches based on the information about earlier branch outcomes stored in the table. Unfortunately, these tables are a scarce resource, and middleware applications can be so large that by the time a branch instruction is fetched a second or subsequent time, the history for that instruction has already been ejected. When there is no historical data in the table

for a branch, the processor falls back to the default not-taken prediction for a forward branch. The processor then speculatively executes the tracing code only to discover several cycles later that it had mispredicted the branch and must throw away all the work it has done. Recovering from a branch misprediction can take more than 10 processor cycles, depending on the processor, plus any work it has done since the prediction was made must also be discarded.

The third problem results in instruction cache and, potentially, TLB misses because the code executing is typically spread over more cache lines or memory pages than is necessary.

To solve these three problems, the JIT reorders the basic blocks of each method to make each block's most commonly executed successor its fall-through successor. Since the JIT has information about which direction each branch favors, branches with a relatively biased outcome such as debugging tests can be flipped so that the processor's default not-taken prediction for forward branches becomes the correct prediction. Some processors do not even try to remember a branch if its prediction matches the default prediction, which means this optimization can have a secondary performance benefit because more branches can be maintained in the prediction tables than before. Finally, because the commonly executed code appears close together in memory, fewer instruction cache and TLB misses are expected. Although reordering code along hot paths is not a new idea [13], we have found it to be particularly effective in the context of a JIT compiler.

5. Results

In this section, we report the benefits we have seen in a group of benchmarks relevant to server and middleware application performance. The improvements we report have not been collected at a single point in time but rather over the course of the recent product development cycle for the practical reason that it is not always possible or reasonable to maintain code to selectively enable or disable particular features. In a production JIT compiler, wherever serviceability is not improved by such code, it is eliminated after a testing cycle to simplify code maintenance.

The impact of this practise on the results of this section is that individual improvements should not be considered additive. If two improvements are measured at 3% and 5%, one should not conclude that the two improvements together would provide an aggregate 8% speedup; the two measurements have implicitly different baselines. Furthermore, IBM maintains more than one Java JIT product and some improvements are specific to one particular technology base. The two products we have evaluated are the IBM Developer Kit for Java and the J9 Java virtual machine.

Moreover, some of the improvements described in this section are platform-specific. These results should not be carried across to other platforms as the improvements will vary considerably from platform to platform; the work done for one platform may be irrelevant or even harmful for another. To the best of our knowledge, we have not excluded any result available to us simply because it degraded performance.

In all cases, we have made an effort to collect together improvements that can reasonably be considered together by platform or by underlying JIT technology, though these improvements should never be interpreted in any additive sense except where explicitly noted.

The results in this section are organized by benchmark. The improvements that had an impact on each benchmark are listed in each subsection. The benchmarks we discuss are: SPECjbb2000[16], SPECjAppServer2002[15], XML Parser[14], a cryptography-based micro-benchmark, and IBM WebSphere® Application Server start-up.

5.1. SPECjbb2000

The SPECjbb2000 benchmark models a pure server application without using an application server or database tier. The benchmark encodes the logic and database for a business managing orders, inventory, deliveries, and payments. It is inspired by, but not directly comparable to, the TPC-C database workload. The benchmark progressively increases its workload to test the throughput supported by a particular system by increasing the number of operating warehouses over time, starting at 1 warehouse and ending at 2N warehouses where N is usually the number of processors in the machine. The results reported below are improvements to the final score, which is an aggregate metric that primarily tracks the average of the throughputs achieved at warehouses P through 2P, where P is the warehouse at which the peak throughput was measured (usually P = N).

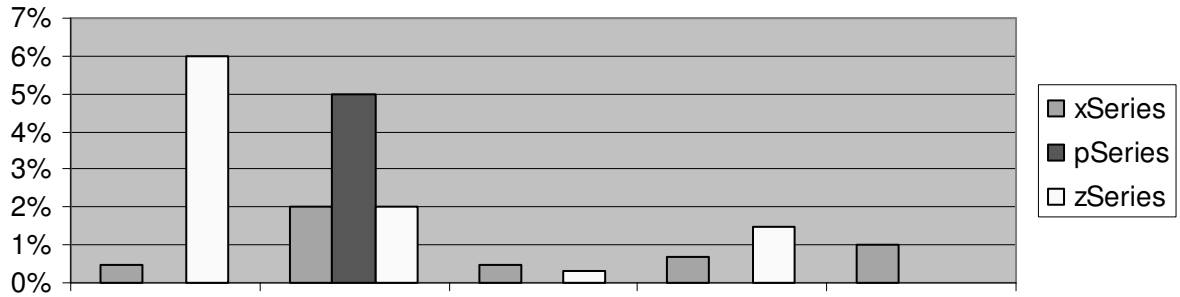

Figure 1: SPECjbb2000 Improvements, IBM Developer Kit for Java

Many of the features we've described in Section 3 improve the performance of the SPECjbb2000 benchmark. The benefit for each such feature for three platforms (xSeries®, pSeries®, and zSeries®) and for the IBM Developer Kit for Java product is shown in Figure 1. The percentage improvement is in the self-reporting score relative to the same score without the specific feature. Note that some bars are not present in this graph because those features have not been implemented for that particular platform or for this particular product. The xSeries results were measured on a 4x2.8Ghz Pentium 4 system with Intel Hyper-threading enabled running Microsoft Windows 2000. The pSeries results were measured on a 16-way 1.1Ghz POWER4 p670 system running AIX® 5.1. The zSeries results were measured on a variety of different machine configurations and so these results in particular should not be compared directly against one another.

For the J9 virtual machine product, we report the improvements found in Figure 2. Again, some bars are not present because that feature was not implemented for a particular platform or in this product, or because that feature had no benefit on that particular platform.

5.2. SPECjAppServer2002

The SPECjAppServer2002 benchmark is a large-scale middleware application benchmark that models the complete business processes of a Fortune-500 company, from customer ordering and invoicing through inventory and supply chain management to manufacturing. The benchmark exercises both an application server and a database tier, which can reside on the same machine, on different machines, or each tier can be distributed across many machines. It is a strenuous test of Enterprise JavaBeans (EJB) 2.0. From the JIT's perspective, our main ability to improve this benchmark is within the application server, where a very flat and widely distributed profile greatly hinders the identification of "bang-for-the-buck" performance improvements.

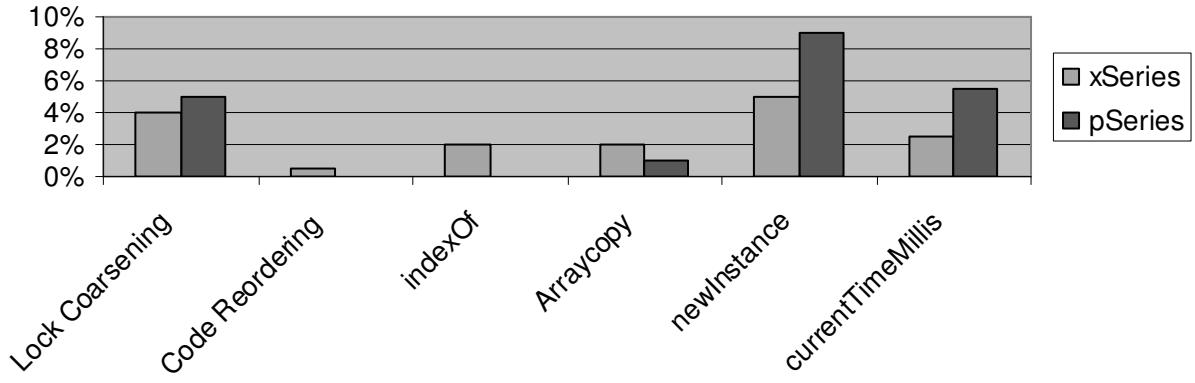

Figure 2: SPECjbb2000 Improvements, J9 virtual machine

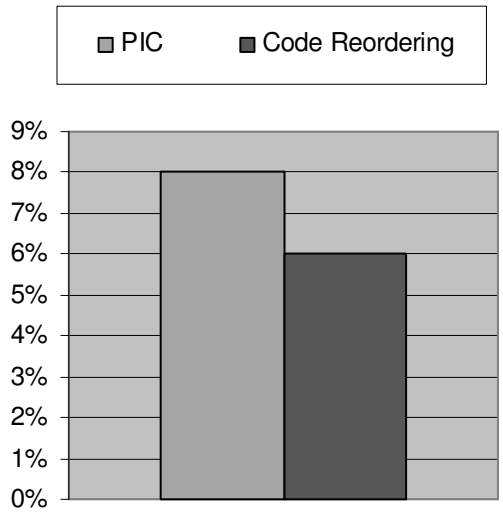

Figure 3: SPECjAppServer2002 Improvements

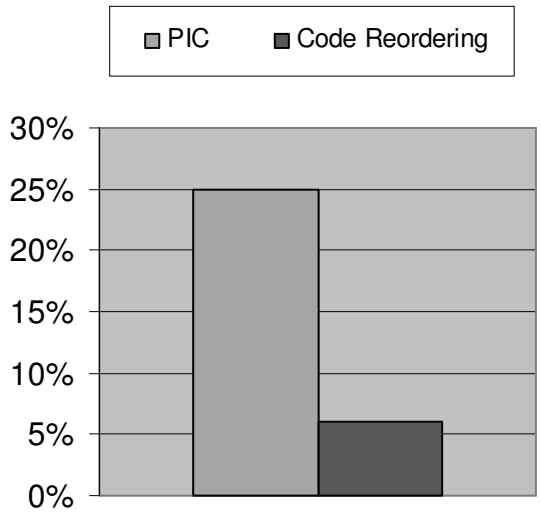

Figure 4: XML Parser Benchmark Improvements

Our investigation of the SPECjAppServer2002 benchmark is still in its preliminary stages. Nevertheless, we have observed that two of the improvements discussed in Section 4 have significantly improved the performance of this benchmark: polymorphic inline caches and code reordering (see Figure 3).

In this particular case, these two measurements can be combined since the improvement we have seen by enabling both of these features in the IBM Developer Kit for Java was 14%. In the context of the incredibly flat profile for this application, both results are even more impressive. We are continuing to investigate improving the performance of this benchmark. As for the SPECjbb2000 benchmark, the percentage improvement is in the self-reporting score relative to the same score without the specific feature. The performance improvements were measured on an xSeries 4x2.8Ghz Pentium 4 system with Hyper-threading enabled running Windows 2000.

5.3. XML Parser

The XML Parser benchmark we have used is based on the Apache Xerces XML Parser for Java [14] that tests parser performance on different combinations of XML data sets. One of the main design goals for the Xerces parser is extensibility; the implementation of the parser relies heavily on abstract classes and method invocations via interface classes. An example of such a class is `AbstractSAXParser`. Given this design approach, proper implementation of Polymorphic Inline Caches (PICs) can yield significant improvements as measured on an xSeries 1.6 Ghz Pentium 4 uniprocessor running Microsoft Windows 2000, which are presented in Figure 4, measured on the IBM Developer Kit for Java product. Note that the scale of the y-axis is different than that of the adjacent Figure 3. We tested the XML Parser benchmark using several different sample XML files with sizes ranging from 1 KB to 5 MB. The benchmark parses each of the files a number of times and calculates a ratio score reflecting the number of elements parsed per second for each data set. The benchmark also computes an average score by combining and interleaving the parsing for each sample XML file into a single run. Figure 4 also includes the improvement in the parser throughput because of the code reordering technique described in Section 4.4.

5.4. Cryptography

The cryptography benchmark we used for our experiments is a micro-benchmark consisting of a loop that performs extended precision integer operations. The loop computes A*B+C where A and C are 1024-bit precision integers simulated by vectors of 32 unsigned 32-bit values using arrays of the Java long data type[2] and B is a 32-bit unsigned number. The loop body contains one long addition and one long multiplication operation. On a 32-bit

[2] Java has no unsigned integer types, so longs are typically used to hold unsigned 32-bit values.

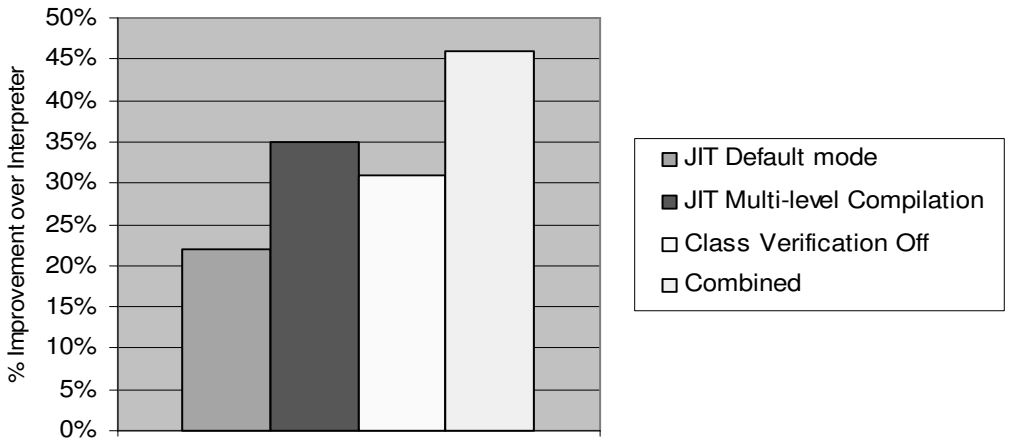

Figure 5: Application Server Start-up Improvements

platform, the long add operation naïvely expands to two 32-bit operations and four 32-bit loads. Similarly, the long multiplication, depending on the instruction set architecture (ISA) of the platform, naïvely requires three 32-bit multiplies (two 32-bit by 32-bit producing low order 32-bit results and one full 32-bit by 32-bit multiply with 64-bit result), two adds to compute the high order 32-bits of the product from the three multiplies, and four 32-bit loads.

The JIT compiler is able to discover that the long operands are actually 32-bit unsigned quantities and therefore that the high order 32-bits of each long used to hold an unsigned 32-bit value are provably zero. This discovery enables the JIT to dramatically simplify the computation and memory traffic that occurs in computing this inner product. After the JIT recognizes the 32-bit unsigned multiplication, the computation is performed by one 32-bit by 32-bit multiply with 64-bit product, no adds, and two loads. Similarly, the 64-bit add operation in the loop is reduced from two adds and four loads to two adds and two loads.

Transforming these computations as described above results in an almost 50% performance improvement, which, in the case of X86, is due to the reduced computation and operand driven memory traffic as well as the complete elimination of spill and fill instructions in the loop. The results are based on measurements performed on an xSeries 1.6 GHz Pentium 4 uniprocessor running Microsoft Windows 2000.

5.5. Application Server Start-up

As the basis for application server start-up performance analysis we chose the IBM WebSphere Application Server product, where we report the levels of improvement that can be achieved by applying multi-level optimization recompilation strategies. Since significant class loading occurs at server start-up, we show that significant additional performance improvement can be achieved by reducing the class loading cost by turning off class verification. Figure 5 shows the percentage improvements relative to the time taken by the interpreter to boot the application server to its ready state. The results were collected by running the IBM WebSphere Application Server product with the J9 virtual machine on an xSeries 1.6 GHz Pentium 4 uniprocessor on Microsoft Windows 2000.

6. Summary

In this paper, we have made three basic contributions. 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 described 12 different optimizations and features that our teams have developed recently to improve the performance of both server and middleware applications. Third, we presented results that show the benefits of robust implementations of these optimizations for a variety of applications. These results demonstrate both the effectiveness of the features we have implemented as well as the level of improvements that can reasonably be expected for robust implementations in a high performance production JIT.

Acknowledgements

The authors would like to thank the members of the IBM Java JIT and VM development teams whose work is described in this paper. In particular, we

would like to acknowledge the feedback and advice from Alan Adamson, Mike Fulton, and Derek Inglis in the preparation of this paper. We also thank the reviewers for their time and effort to evaluate this paper.

References

[1] J. Aldrich, E. G. Sirer, C. Chambers, and S. Eggars. *Comprehensive Synchronization Elimination for Java*. In Science of Computer Programming, Volumn 47, Issue 2-3, May 2003.

[2] B. Alpern, A. Cocchi, S. Fink, D. Grove, and Derek Lieber. *Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless*. In ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Oct 2001.

[3] J. Bogda and U. Holzle. *Removing Unnecessary Synchronization in Java*. In Conference on Object-Oriented Programming, Systems, Languages, and Applications, Nov 1999.

[4] R. Boyer and S. Moore, *A Fast String Searching Algorithm*, CACM 20(10), pg 762-772 (1977).

[5] B. Calder, P. Feller, and A. Eustace. *Value Profiling and Optimization*. Journal of Instruction Level Parallelism, Mar. 1999.

[6] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. *Escape Analysis for Java*. In ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. Denver, Colorado. Nov. 1999.

[7] P. Diniz and M. Rinard. *Lock Coarsening: Eliminating lock overhead in automatically parallelized object-based programs*. In Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing. San Jose, CA, Aug 1996.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[9] 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.

[10] U. Holzle, C. Chambers, and D. Ungar. *Optimizing dynamically-typed object-oriented languages with polymorphic inline caches*. In P. America, editor, Proceedings ECOOP'91, LNCS 512, pages 21-38, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification Second Edition*. The Java Series. Addison-Wesley, 1999.

[13] S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers. 1997.

[14] The Apache XML Project, 2001. http://xml.apache.org/xerces-j

[15] The Standard Performance Evaluation Corporation. SPECjAppServer 2002 Benchmark. http://www.spec.org/jAppServer2002, 2002.

[16] The Standard Performance Evaluation Corporation. SPEC JBB 2000 Benchmark. http://www.spec.org/osg/jbb2000, 2000.

[17] J. Vitek, and R. N. Horspool. *Compact dispatch tables for dynamically typed object oriented languages*. In Proceedings of International Conference on Compiler Construction (CC'96) pages 281-293, Apr. 1996. Published as LNCS, vol 1060.

[18] P. Wilson. *Uniprocessor Garbage Collection Techniques*. In Proceedings of International Workshop on Memory Management. Springer-Verlag. Saint-Malo, France. 1992.

Trademarks