Impact of JIT/JVM Optimizations on Java Application Performance

K. Shiv[+], R. Iyer[+], C. Newburn[+], J. Dahlstedt[*], M. Lagergren[*] and O. Lindholm[*]
[+]Intel Corporation
[*]BEA Systems

Abstract

With the promise of machine independence and efficient portability, JAVA has gained widespread popularity in the industry. Along with this promise comes the need for designing an efficient runtime environment that can provide high-end performance for Java-based applications. In other words, the performance of Java applications depends heavily on the design and optimization of the Java Virtual Machine (JVM). In this paper, we start by evaluating the performance of a Java server application (SPECjbb2000^{TM}) on an Intel platform running a rudimentary JVM. We present a measurement-based methodology for identifying areas of potential improvement and subsequently evaluating the effect of JVM optimizations and other platform optimizations. The compiler optimizations presented and discussed in this paper include peephole optimizations and Java specific optimizations. In addition, we also study the effect of optimizing the garbage collection mechanism and the effect of improved locking strategies. The identification and analysis of these optimizations are guided by the detailed knowledge of the micro-architecture and the use of performance measurement and profiling tools (EMON and VTune) on Intel platforms.

1 Introduction

The performance of Java client/server applications has been the topic of significant interest in the recent years. The attraction that Java offers is the promise of portability across all hardware platforms. This is accomplished by using a managed runtime engine called the Java Virtual Machine (JVM) that runs a machine-independent representation of Java applications called bytecodes. The most common mode of application execution is based on a Just-In-Time (JIT) compiler that compiles the bytecodes into native machine instructions. These native machine instructions are also cached in order to allow for fast re-use of the frequently executed code sequences. Apart from the JIT compilation, the JVM also performs several functions including thread management and garbage collection. This brings us to the reason for our study i.e. Java application performance depends very heavily on the efficient execution of the Java Virtual Machine (JVM). Our goal in this paper is to characterize, optimize and evaluate a JVM while running a representative Java application.

Over the last few years, several projects (from academia as well as in the industry) [1,2,4,7,8,9,10,15,16,21] have studied various aspects of Java applications, compilers and interpreters. We found that R. Radhakrishnan et al. [16] cover a brief description of much of the recent work on this subject. In addition, they also provide insights on architectural implications of Java client workloads based on SPECjvm98 [18]. Overall, the published work can be classified into the following general areas of focus: (1) presenting the design of a compiler, JVM or interpreter, (2) optimizing a certain aspect of Java code execution, and (3) discussing the application performance and architectural characterization. In this paper, we take a somewhat different approach touching upon all the three aspects listed above. We present the software architecture of a commercial JVM, identify several optimizations and characterize the performance of a representative Java server benchmark through several phases of code generation optimizations carried out on a JVM.

Our contributions in this paper are as follows. We start by characterizing SPECjbb2000 [17] performance on Intel platforms running an early version of BEA's JRockit JVM [3]. We then identify various possible optimizations (borrowing ideas from literature wherever possible), present the implementation details of these optimizations in the JVM and analyze the effect of each optimization on the execution characteristics and overall performance. Our performance characterization and evaluation methodology is based on hardware measurements on Intel platforms - using performance counters (EMON) and a sophisticated profiler (VTune [11]) that allows us to characterize various regions of software execution. The code generation enhancements that we implement and evaluate include (1) code quality improvements such as peephole optimizations, (2) dynamic code optimizations, (3) parallel garbage collection and (4) fine-grained locks. The outcome of our work is the detailed analysis and breakdown of benefits based on these individual optimizations added to the JVM.

The rest of this paper is organized as follows. Section 2 covers a detailed overview of the BEA JRockit JVM, the measurement-based characterization methodology and the SPECjbb2000 benchmark. Section 3 discusses the opti-

mizations - how they were identified, implemented and their performance evaluation. Section 4 summarizes the breakdown of the performance benefits and where they came from. Section 5 concludes this paper with some direction on future work in this area.

2 Background and Methodology

In this section, we present a detailed overview of JRockit (the commercial JVM used) [3], SPECjbb2000 (the Java server benchmark) [17] and the optimization and performance evaluation methodology and tools.

2.1 Architecture of the JRockit JVM

The goal of the JRockit project is to build a fast and efficient JVM for server applications. The virtual machine should be made as platform independent as possible without sacrificing platform specific advantages. Some of the considerations included reliability, scalability, non-disruptiveness and of course, high performance.

JRockit starts up differently from most ordinary Java JVMs by first JIT-compiling the methods it encounters during startup. When the Java application is running, JRockit has a bottleneck detector active in the background to collect runtime statistics. If a method is executed frequently and found to be a bottleneck, it is sent to the Optimization Manager subsystem for aggressive optimization. The old method is replaced by the optimized one while the program is running. In this way, JRockit is using adaptive optimization to improve code performance. JRockit relies upon a fast JIT-compiler for unoptimized methods, as opposed to interpretative byte-code execution. Other JVMs such as Jalapeno/Jikes [23] have used similar approaches.

It is important to optimize the garbage collection mechanism in any JVM in order to avoid disruption and provide maximum performance to the Java application. JRockit provides several alternatives for garbage collection. The ”parallel collector” utilizes all available processors on the host computer when doing a garbage collection. This means that the garbage collector runs on all processors, but not concurrently with the Java program. JRockit also has a concurrent collector which is designed to run without ”stopping the world”, if non-disruptiveness is the most important factor.

To complete the server side design, JRockit also contains an advanced thread model, that makes it possible to run several thousands of Java threads as light weight tasks in a very scalable fashion.

2.2 Overview of the SPECjbb2000 Benchmark

SPECjbb2000 is Java Business Benchmark from SPEC that evaluates the performance of Server Side Java. It emulates a three-tier system, with business logic and object manipulation, the work of the middle layer predominating.

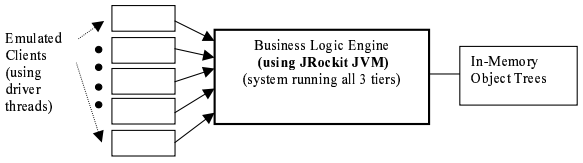

Figure 1. The SPECjbb2000 Benchmark Process

The database component requirement common to three-tier workloads is emulated using binary trees of objects. The clients are similarly replaced by driver threads. Thus, the whole benchmark runs on a single computer system, and all the three tiers run within the same JVM. The benchmark process is illustrated in Figure 1.

The SPECjbb2000 application is somewhat loosely based on the TPC-C [20] specification for its schema, input generation, and operation profile. However, the SPECjbb2000 benchmark only stresses the server-side Java execution of incoming requests and replaces all database tables with Java classes and all data records with Java objects. Unlike TPC-C where the database execution requires disk I/O to retrieve tables, in SPECjbb2000 disk I/O is completely avoided by holding the objects in memory. Since users do not reside on external client systems, there is no network IO in SPECjbb2000 [17].

SPECjbb2000 measures the throughput of the underlying Java platform, which is the rate at which business operations are performed per second. A full benchmark run consists of a sequence of measurement points with an increasing number of warehouses (and thus an increasing number of threads), and each measurement point is work done during a 2-minute run at a given number of warehouses. The number of warehouses is increased from 1 until at least 8. The throughputs for all the points from N warehouses to 2*N inclusive warehouses are averaged, where N is the number of warehouses with best performance. This average is the SPECjbb2000 metric.

2.3 Performance Optimization and Evaluation Methodology

The approach that we have taken is evolutionary. Beginning with an early version of JRockit, performance was analyzed and potential improvements were identified. Appropriate changes were made to the JVM and the new version of the JVM was then tested to verify that the modifications did deliver the expected improvements. The new version of the JVM was then analyzed in its turn for the next stage of performance optimizations. The types of performance optimizations that we investigated were two-fold. Changes were made to the JIT so that the quality of the generated code was superior, and changes were made to other parts of the JVM, particularly to the Garbage Collector, Object Allo-

cator and synchronization, to enhance the processor scaling of the system.

Our experiments were conducted on a 4 processor, 1.6 GHz, Xeon platform with 4GB of memory. The processors had a 1M level-3 cache along with a 256K level-2 cache. The processors accessed memory through a shared 100 MHz, quad-pumped, front side bus. The network and disk I/O components of our system were not relevant to studying the performance of SPECjbb2000, since this benchmark does not require any I/O. Several performance tools assisted us in our experiments. Perfmon, a tool supplied with Microsoft's operating systems, was useful in identifying problems at a higher level, and allowed us to look at processor utilization patterns, context switch rates, frequency of system calls and so on. EMON gave us insight into the impact of the workload on the underlying micro-architecture and into the types of processor stalls that were occurring, and that we could target for optimizations. VTune permitted us to dig deeper by identifying precisely the regions of the code where various processor micro-architecture events were happening. This tool was also used to study the generated assembly code. The next section describes the performance tools – EMON and VTune – in some more detail.

2.4 Overview of Performance Tools - EMON and VTune

This section describes the rich set of event monitoring facilities available in many of Intel's processors, commonly called EMON, and a powerful performance analysis tool based on those facilities, called VTune [11].

2.4.1 EMON Hardware and Events Used

The event monitoring hardware provides several facilities including simple event counting, time-based sampling, event sampling and branch tracing. A detailed explanation of these techniques is not within the scope of this paper. Some of th key EMON events leveraged in our performance analysis include (1) Instructions – the number of instructions architecturally retired, (2) Unhalted cycles – the number of processor cycles that the application took to execute, not counting when that processor was halted, (3) Branches – the number of branches architecturally retired which are useful for noting reductions in branches due to optimizations, (4) Branch Mispredictions – the number of branches that experienced a performance penalty on the order of 50 clocks, due to a misprediction, (5)Locks – the number of locked cmpxchg instructions, or instructions with a lock prefix and (6) Cache misses – the number of misses and its breakdown at each level of the cache hierarchy.

The reader is referred to the Pentium 4 Processor Optimization Guide [24] for more details on these events.

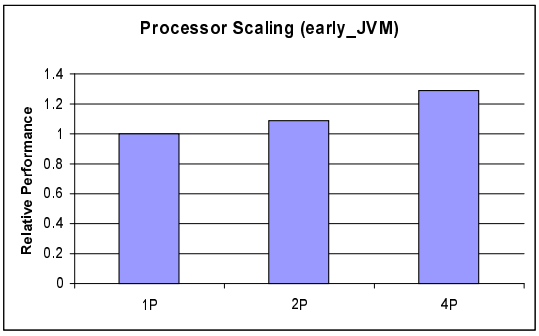

Figure 2. Processor Scaling on an early JRockit JVM version

2.4.2 VTune Performance Monitoring Tool

Intel's VTune performance tools provide a rich set of features to aid in performance analysis and tuning: (1) Time-based and event-based sampling, (2) Attribution of events to code locations, viewed in source and/or assembly, (3) Call graph analysis and (4) Hot spot analysis with the AHA tool, which indicates how the measured ranges of event count values compare with other applications, and which provides some guidance on what other events to collect and how to address common performance issues. One of the key tools provides the means for providing the percentage contribution of a small instruction address range to the overall program performance, and for highlighting differences in performance among versions of applications and different hardware platforms.

3 JVM Optimizations and Performance Impact

In this section we describe the various JVM improvements that we studied and document their impact on performance. We also show the analysis of JVM behavior and the identification of performance inhibitors that informed the improvements that were made.

3.1 Performance Characteristics of an early JVM

The version of JRockit with which we began our experiments was a complete JVM in the sense that all of the required JVM components were functional. Unlike several other commercial JVMs though, JRockit does not include an interpreter. Instead, all application code is compiled before execution. This could slow down the start of an application slightly, but this approach enables greater performance. JRockit also included a selection of Garbage Collectors and two threading models.

Figure 3 shows the performance for increasing numbers of warehouses for a 1-processor and a 4-processor system.

Metric of Interest	1P Avg	1P Max	4P Avg	4P Max
Total Processor %	94.71	100.00	56.25	100.00
Total Privileged %	0.85	13.44	3.62	7.58
Total User %	93.85	100.00	52.63	100.00
Total Interrupt%	0	0	0	0
JRockit Thread Count	8	8	18	18
System Calls/sec	340	4686	55100	92530
Context Switches/sec	98	358	20991	23945
Interrupts/sec	93	191	279	382
Processor Queue Length	5.27	9.00	1.14	5.00

Table 1. System Performance Characteristics for early_JVM

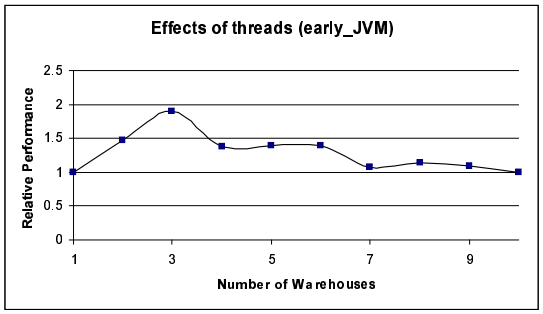

Figure 3. Performance Scaling with Increasing Warehouses

There is a marked roll-off in performance from the peak at 3 warehouses in the 4-processor case. The JVM can thus be seen to be having some difficulty with increasing numbers of threads.

Data obtained using Perfmon is shown in Table 1. While the utilization of the 1 processor is quite good at 94%, the processor utilization in the 4-processor case is only 56%. It is clear that improvements are needed to increase the processor utilization. The context switch and system call rates are two orders of magnitude larger in the 4P than in the 1P. The small processor queue length indicates the absence of work in the system. These aspects along with the sharp performance roll-off with increased threads, all point to a probable issue related to synchronization. It appears likely that one or more locks are being highly contended, resulting in a large number of the threads being in a state of suspension waiting for the lock.

While being fully functional, this version of JRockit (we call it early_JVM) had not been optimized for performance. It thus served as an excellent test-bed for our studies. The processor scaling seen with the initial, non-optimized early_JVM is shown in Figure 2. It is obvious that we can do much better on scaling. Many other statically compiled workloads exhibit scaling of 3X or better from 1 processor to 4 processors, for instance.

3.2 Granularity of Heap Locks

The early version of JRockit performed almost all object allocation globally with all allocating threads increasing a pointer atomically to allocate. In order to avoid this contention, thread local allocation and Thread Local Areas (TLAs) were introduced. In that scheme, each thread has its own TLA to allocate from and the atomic operation for increasing the current pointer could be removed, only the allocation of TLAs required synchronizations.

A chain is never stronger than its weakest link, once a contention on a lock or an atomic operation is removed, the problem usually pops up somewhere else. The next problem to solve was the allocation of the TLAs. For each TLA that was allocated, the allocating thread had to take a ”heap lock”, find a large enough block on a free list and release the lock. The phase of object allocation that requires space to be allocated from a free list requires a lock. This lock acquisition and release showed up on all our measurements with VTune as a hot spot, marking it as a highly contended lock.

One attempt was made to reduce the contention of this lock by letting the allocating thread allocate a couple of TLAs and putting them in a smaller temporary storage where they could be allocated using only atomic operations by other threads. This attempt was a dead end. Even if the thread that had the heap lock put a large amount of TLAs in the temporary storage, all threads still ended up waiting most of the time, either for the heap lock or that the holder of the heap lock would give away TLAs.

The final solution was to create several TLA free lists. Each thread has a randomly allotted ”home” free list from which to allocate the TLAs it needs. If the chosen list was empty, the allocating thread tried to take the heap lock and fill that particular free list with several TLAs. After this, the thread would choose another ”home” free list randomly to allocate from. By having several lists, usually only one thread would try to take the heap lock at the same time and the contention of the heap lock was reduced dramatically. Contention was further reduced by providing a TLA cache; the thread that acquires the heap lock moves 1MB of memory into the cache. A thread that finds its TLA free list empty checks for TLAs in the cache before taking the heap lock.

Figure 4 shows the marked improvement in processor scaling in the modified JVM, the JVM with the heap lock contention reduction. Scaling at 2 processors has increased from 1.08X to 1.70X, and the scaling at 4 processors has improved to 2.46X from 1.29X. The perfmon data with these changes is interesting, and is shown in Table 2. The increase in processor utilization and the decrease in system calls and context switches are all very dramatic.

3.3 Garbage Collection Optimizations

The early version of JRockit included both a single and multi generational concurrent garbage collector, designed to

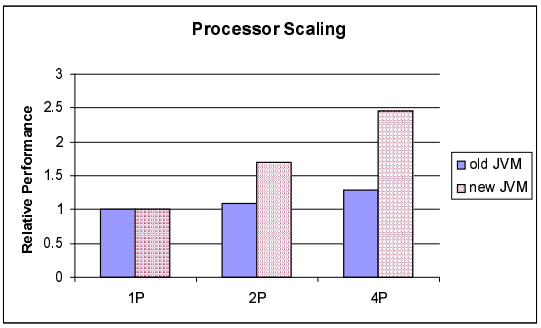

Figure 4. Improvement of Processor Scaling with Heap Lock contention scaling

Metrics of Interest	old JVM (avg)	old JVM (max)	new JVM (avg)	new JVM (max)
Total Processor %	56.25	100.00	81.95	87.50
Total Privileged %	3.62	7.58	0.30	5.70
Total User %	52.63	100.00	81.64	100.00
Total Interrupt%	0	0	0	0
JRockit Thread Count	18	18	10	10
System Calls/sec	55100	92530	470	1853
Context Switches/sec	20991	23945	70	1151
Interrupts/sec	279	382	263	391
Processor Queue Length	1.14	5.00	1.14	3.00

Table 2. System Performance Characteristics after Heap Lock Improvements

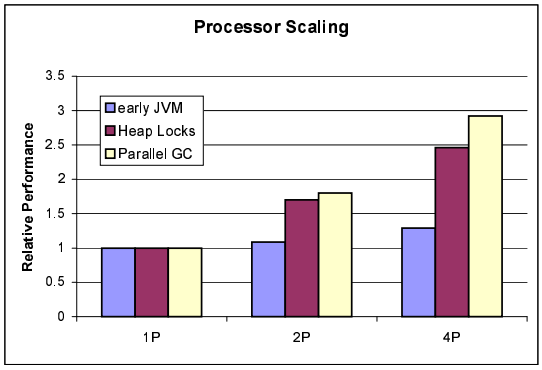

Figure 5. Impact of Parallel Garbage Collection on Processor Scaling

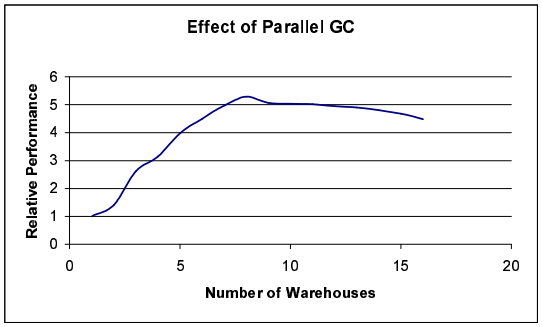

Figure 6. Impact of Parallel Garbage Collection on SPECjbb2000 Performance

have really short pause times and fair throughput. Throughput in a concurrent collector is usually not a problem since a full collection is rarely noticed, even less on a multiprocessor system. The problem occurs when objects are allocated in such a fast rate that even if the garbage collector collects all the time on one processor and lets the other processors run the program, the collector still doesn't manage to keep up the pace. This problem started to hurt performance badly in JRockit when running 8 warehouses on 8-way systems.

To solve this, the so-called "parallel collector" was developed. The base was a normal Mark and Sweep [13] collector with one marking thread per processor. Each thread had its own marking stack, and if a stack is empty the thread could work-steal references from other stacks [5]. Normal pushing and popping required no synchronization or atomic operations, only the work-stealing required one atomic operation. Each thread also had an expandable local stack to handle overflow in the exposed marking stack.

Sweeping is also done in parallel by splitting the heap in N sections and letting each thread allocate a section, sweep it, allocate a new section and so forth until all sections were swept. The sweeping algorithm focused on performance more than accuracy, creating room for fragmentation if we were unlucky. A partial compaction scheme was employed to reduce this fragmentation.

These GC optimizations resulted in an increase in the reported SPECjbb2000 result in a 4P system, and improved processor scaling from 2.46 to 2.92, as illustrated in Figure 5. The benefits of this were more noticeable at higher numbers of warehouses and therefore lead to a much flatter roll-off from the peak, as shown in Figure 6.

3.4 Code Quality Improvements

Several code quality improvements were made during the benchmarking process. A new code generation pipeline was developed and merged into the product. This enabled us to do a lot more versatile and low-level optimizations on code than previously was possible. Based on the

SPECjbb2000 characteristics measured and analyzed in the previous section, we were able to identify several patterns at the native code level that were suboptimal. The JRockit team replaced these with better code through peephole optimizations (commonly used for compiler optimizations as in [6, 14]) or more efficient code generation methodologies. While the compiler optimizations listed below are well-known and understood, the requirement here is

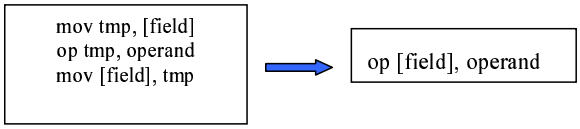

Figure 7. A Simple Example of Peep-Hole Optimization

that the compile time overhead be kept to a minimum since it is a part of the execution time; and as such, not all known optimizations and techniques could be added. These are by no means a complete list of improvements, but give some perspective on things that were done to enhance code quality.

1. **Peephole Optimizations:** The new JRockit code generator made it possible to work with native code just before emission, i.e. there would be IR operations for each native code operation. Several small peephole optimizations were implemented on this. We present one example of this kind of pattern matching here: Java contains a lot of load/store patterns, where a field is loaded from memory, modified and then rewritten. Literal translation of a Java getfield/putfield sequence would result in three instructions on IA32 as shown in Figure 7(left). IA32 allows most operations to operate directly on addresses, so the above sequence could be collapsed to a single instruction as shown in Figure 7(right).

2. **Better use of IA32 FPU instructions:** Java has precise floating-point semantics, and works either in 32-bit or 64-bit precision. This is usually a problem if one wants to use fast 80-bit floating points that there is hardware support for on IA32, but in some cases we don't need fp-strict calculations and can use built in FPU instructions. JRockit was modified to determine when this is possible.

3. **Better SSA reverse transform:** Most code optimizations take place in SSA form. There were some problems with artifacts in the form of useless copies not being removed from the code when transforming back to normal form. The transform was modified to get rid of these, with good results. Register pressure dropped significantly for optimized code.

4. **Faster checks:** The implementation of several Java runtime checks was speeded up. Some Java runtime checks are quite complicated, such as the non-trivial case of an array store check. These were treated as special native calls, but without using all available registers. Special interference information for these simplified methods was passed to the register allocator, enabling less saves and restores of volatile registers.

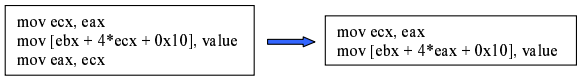

Figure 8. An Example of Copy Propagation

Code Generator Improvements			
Metric of Interest	Before CGI	After CGI	% Change
Norm. Perf	1	1.34	34%
CPI	2.65	2.67	-1%
Path Length	57940	42128	27%

Table 3. Impact of Better Code Generation on Application Performance

5. **Specializations for common operations:** Array allocation was re-implemented with specialized allocation policies for individual array element sizes. The Java "arraycopy" function was also specialized, depending on if it was operating on primitives or references and on elements of specific sizes. Other common operations were also specialized.

6. **Better Copy Propagation:** The copy propagation algorithm was improved and also changed to work on the new low level IR, with all its addressing modes and operations. An example of better copy propagation is shown in Figure 8.

These improvements to the JIT were undertaken to reduce the code required to execute an application. It is possible that the techniques used to lower the path length could increase the CPI of the workload, and end up hurting throughput. One example of this would be the usage of a complex instruction to replace a set of simpler instructions. However, Table 3 shows that while the efforts to reduce the path length were well rewarded with a 27% improvement for SPECjbb2000, these optimizations did not hurt the CPI in any significant way. The path length improvement resulted in a 34% boost to the reported SPECjbb2000 result.

3.5 Dynamic Optimization

The initial compile time that is tolerable limits the extent to which compiler optimizations can be applied. This implies that while JRockit provides better code in general than an interpreter, for the few functions that other JITs do choose to compile, there is a risk of under-performance. JRockit has chosen to handle this issue by providing a secondary compilation phase that can include more sophisticated optimizations, and using this secondary compilation during the application run to compile a few frequently used hot functions.

There are two main issues with this that impact performance. Since we cut into the application run-time when we re-optimize code, it is essential to ensure that the hottest functions are targeted, so that the performance benefit is worthwhile. However, the method used to identify the best target functions must not be very intrusive and cut into performance.

Early_JVM, our baseline JRockit, used Method Invocation Counters (MIC) to keep track of how often a function is called. Every time a function called another function, the MIC of both functions were incremented - while it is obvious as to why the callee MIC was incremented, the purpose of incrementing the caller's MIC is to be able to catch the optimization at the root functions rather than just at the leaves. While running multi-threaded code, it is of course possible that the same function could be executed on two separate processors simultaneously. Both processors would then need to attempt to increment the function's MIC. Locking the increment operation would slow it down and make it too intrusive, so the design decision was taken to not do so. However, even otherwise, there is a significant performance impact of transferring the counter from cache to cache, thus decreasing cache performance and increasing bus traffic.

Every 3 seconds, JRockit would check the MIC and look for a target function to optimize. To avoid taking away too much time from the application, only one function is targeted at each time. Finding the exact function with the highest MIC could also get very expensive. SPECjbb2000, for example, has more than 2000 functions. To frequently find the maximum of 2000 values would be significantly intrusive. Instead JRockit scans through the list of functions looking for one whose MIC is higher than a threshold, targets that for optimization, and sets its MIC to zero.

JRockit provided a command line option (-Xnoopt) with which the secondary compilation can be turned off. JRockit also provided a command line option (-XXoptall originally, later removed) that could be used to force the optimization of all functions at the initial code generation phase itself. Using just the -XXoptall option does not turn off the secondary compilation phase. Since the performance improvements due to re-optimization when the original code was already optimized is small, running with -XXoptall and comparing the results with -XXoptall -Xnoopt, allows us to get a measure of the intrusiveness of the secondary compilation phase. Similarly comparing a run with neither of these two parameters with a run that includes -Xnoopt gives a measure of the benefits of secondary optimized compilation. We also used VTune to identify the top hot spots in SPECjbb2000, and compared that with the list of functions that were targeted for recompilation, with a view towards studying the efficiency of the hot spot identification process.

Table 4 summarizes our findings. The observation from the throughput measurements is that this approach does provide a 4% net benefit. However, the technique is intrusive and takes up 6% of performance.

Optimization may well provide the most benefit when the JIT has a large block of code to optimize. However, such large methods tended to be hot spots not because of the number of times they were called (when the MIC would increment) but because of the time spent in the function during the few times that they were called. For instance, a relatively hot method TransactionManager.go is invoked only 136 times during a run with up to 16 warehouses, and is therefore never optimized.

Approach	Sampling	MIC
Performance Benefit due to better code	17%	10%
Performance Loss due to intrusiveness	1%	6%
Net Gain	16%	4%
Performance Gained by Optimizing at Start-up	14%	14%
Percent of execution time spent in optimized functions	60%	71%

Table 4. Impact of Dynamic Optimization Strategies

Based on this data, another approach was pursued. The Method Invocation Counters were abandoned, and instead a sampling thread was introduced. The sampling thread wakes up occasionally and checks some or all of the application's threads. It notes down which methods they are in, and additionally it notes down the information for every function in the thread's calling stack. These counts replace those provided by the Method Invocation Counters.

On the positive side this removes a lot of instructions from the application code space, making for both tighter code and shorter path lengths. On the negative side the counts acquired through sampling may be more prone to error. To counteract this, the sampling technique is more likely to correctly represent a method like TransactionManager.go, since it depends more on the time spent in a method than the number of times a method is called.

The data we have obtained is compelling. Dynamically recompiling as few as a 100 functions during the life of the application has produced results that are very comparable to the results produced by optimizing all the methods at the beginning. Using the -Xnoopt flag as the baseline, the -XXoptall provided a benefit of 14%. Using the sampling based optimization, we noted a performance improvement of 16%. It may be noted that this approach can provide a higher benefit than by compiling all methods with optimizations. This is because the functions that are optimized by this method cover more than 70% of execution time, and it is possible to apply optimizations better due to the available history information that -XXoptall lacks.

It is thus possible to have both a relatively quick start-up and excellent performance. Based on these experiments, BEA JRockit has done away with the -XXoptall flag, and Method Invocation Counters. The JVM today uses the sampling approach to identify targets for optimization.

```
// Thin lock:
If (acquire_thin_lock())
        return;
// Stage2lock:
If (not_thin_lock()) {    // checking whether it is a multi-lock
        while ((!acquire_thin_lock()) && (count3<L3)) {
                spin(L1_cycles);
                while((!acquire_thin_lock()) && (count2<L2) {
                        spin(L1_cycles);
                        count2++;
                }
                system_yield();
                count3++;
        }
        wait_and_obtain_thin_lock();
        inflate();
}
```

Figure 9. Algorithm to Implement Fat Lock Deferral

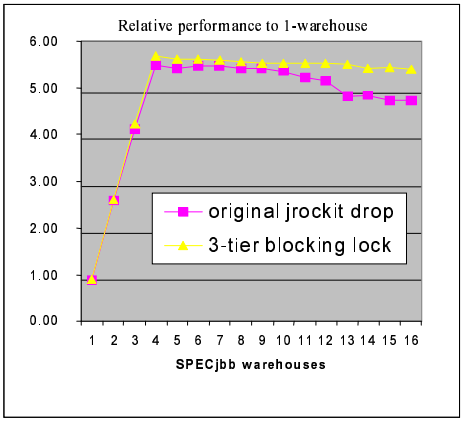

Figure 10. Impact of Fat Lock Deferral

3.6 Lock Inflation Deferral

JRockit defines Java locks as either thin locks or fat locks. All locks are thin locks initially. Whenever any lock is contended for it is inflated to a fat lock. A lock once inflated is never deflated back to a thin lock. Due to the need to access them through extra levels of indirection and hashing, thin locks are much quicker to acquire and release.

Since fat locks are so expensive, it appeared desirable to experiment with approaches to deferring lock inflation. We took as our model the approach implemented by IBM with DB2 [8], which is a three-tier blocking lock algorithm. The algorithm incorporates spinning, yielding and blocking. Sample code to implement this is shown in Figure 9.

The lock is initially requested, and if it is not acquired, the program spins for L1 cycles and tries again. It repeats this sequence of spinning and attempting to lock L2 times before yielding to the Operating System. When the thread awakens it repeats this pattern L3 times, before finally inflating the lock to a fat lock.

We experimented with several values of the parameters L1, L2 and L3, and we found that there was a fairly broad range of values for which the scaling of SPECjbb2000 is much improved. While the specific values are not very interesting (since they differ based on platform configuration), Figure 10 demonstrates how much flatter the curve is, and how much less the performance loss when the number of warehouses is increased. Specifically, our experiments indicate that 99.7% of fat locks are no longer inflated. While the performance impact of this approach is modest in SPECjbb2000, its impact should be much more significant in workloads with higher degrees of contention.

4 Summary of Optimizations, Performance Benefits and Inferences

In the previous section, we discussed the various optimizations that we investigated and incorporated into the JRockit JVM. We also showed the impact of the individual optimizations on the Java application performance using SPECjbb2000 measurements. In this section, we summarize the overall performance improvement made possible through the various phases of optimization.

As shown in Table 5, we basically investigated five different areas of optimization - heap locks, garbage collection, code generation, dynamic optimization and lock deferrals. Of these, the optimizations to heap locks and garbage collection improved processor scaling by a substantial amount. The code generation improvements reduced path length significantly, thereby affecting performance reciprocally. We also demonstrated that it was not necessary to incur slow start-up in the interest of performance as discussed in Section 3.5. In fact, dynamic optimization slightly out-performed pre-optimization. Finally, we also studied the impact of deferring conversion of thin locks to fat locks, showing that it is possible to reduce inflation by as much as 99.7%. While our optimizations focused primarily on path length reduction and processor scaling improvements, future work should center around techniques to improve CPI such as cache conscious object allocation.

5 Conclusions and Future Work

Our aim in this paper was to characterize Java application performance on a JVM as it evolves in its design. We started with a rudimentary, yet commercial, JVM from BEA (JRockit) and studied the performance characteristics of SPECjbb2000 on the JVM. We found that the application had poor scaling characteristics with a multiprocessor (4P vs. 1P) speedup of about 1.3. Upon investigation, we attributed the cause(s) of the poor scaling as well as poor overall performance to a number of potential areas of sub-optimal design. This was done effectively through the means of detailed measurement made possible by performance monitoring tools such as EMON and profiling tools such as VTune.

Areas of Optimization	Performance Characteristic Affected	Performance Benefit Observed
Heap Lock Granularity	Processor Scaling	Improved 1P to 4P performance scaling by 91%
Parallel Garbage Collection	Processor Scaling	Improved 1P to 4P performance scaling by 19%
Code Generation Improvements	Path Length	Reduced path length by 27%
Dynamic Optimization	Application start-up	Throughput increased by 3%
Lock Deferral	Lock Contention	Inflation to fat-locks reduced by 99.7%

Table 5. Summary of Optimizations

Having identified the potential areas for improvement, we tested various schemesand implemented some new optimizations within JRockit. These optimizations included (1) code quality improvements including peephole optimizations, (2) dynamic code generation optimizations, (3) improved garbage collection mechanism and (4) improved locking strategies. In this paper, we showed the impact of each of these optimizations on the performance of the SPECjbb2000 benchmark. The performance improvements gained on the whole were roughly 10X the initial performance of the rudimentary JVM. Apart from the performance gains, we also believe that our measurement-based methodology of performance optimization and tuning will useful to future researchers who embark upon similar studies.

In the future, we will continue to investigate potential JVM optimizations to improve SPECjbb2000 benchmark performance. In addition, we also plan to study upcoming benchmarks that may better represent application servers such as SPECjAppServer2002 [19]. Areas of research would include capturing more profile information through sampling and the usage of that information to generate better code. Since Java code tends to have many small methods and larger code segments may offer better optimization possibilities, investigation into various inlining techniques [2, 12, 22] would be desirable.

Acknowledgements

We would like to thank our colleagues from the MRTE team at Intel Corporation and the JVM development team at BEA Systems for their support during this project.

References

[1] A. Adl-Tabatabai et al., "Fast Effective Code Generation in a Just-in-Time Java Compiler," Proceedings of the ACM SIGPLAN'98 conference on Programming Language Design and Implementation, 1998.

[2] M. Arnold, S. Fink, V. Sarkar, and P. Sweeney, "A comparative study of static and dynamic heuristics for inlining," ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, 2000.

[3] BEA Systems, "Weblogic JRockit: The Server JVM," http://www.bea.com/products/weblogic/jrockit/

[4] A. Barisone, F. Bellotti, R. Berta, and A. De Gloria, "Instruction Level Characterization of Java Virtual Machine Workload," Workload Characterization for Computer System Design, L. John and A. Maynard, eds., pp. 1-24, 1999.

[5] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing", Journal of ACM, 46(5): 720-748, 1999.

[6] J. W. Davidson and D. B. Whalley, "Quick compilers using peephole optimization," Software Practice and Experience, 19(1): 79–97, January 1989.

[7] S. Deickmann and U. Holzle, "A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks," Proc. European Conf. Object Oriented Programming, July 1999.

[8] 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.

[9] M. Gupta, "Optimizing Java Programs: Challenges and Opportunities," Proc. Second Ann. Workshop Hardware Support for Objects and Microarchitectures for Java, Sept. 2000.

[10] 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," Proc. IEEE Compcon '97, pp. 211-216, 1997.

[11] "VTune: Visual Tuning Environment," Available at http://developer.intel.com/design/perftools/vtune/,2002.

[12] S. Jagannathan and A. Wright, "Flow-directed inlining," Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, pp. 193–205, 1996.

[13] R.E. Jones and R. Lins "Garbage Collection: Algorithms for Automatic Dynamic Memory Management," Wiley, July 1996.

[14] S. Montanaro, "A Peephole Optimizer for Python", 7th International Python Conference, Nov 1998.

[15] T. Newhall and B. Miller, "Performance Measurement of Dynamically Compiled Java Executions," Proc. 1999 ACM Java Grande Conference, June 1999.

[16] R. Radhakrishnan, N. Vijaykrishnan, et al., "Java Runtime Systems: Characterization and Architectural Implications," IEEE Transactions on Computers, pages 131-146, vol. 50, issue 2, February, 2001.

[17] "SPEC JBB2000," http://www.spec.org/osg/jbb2000/.

[18] "SPEC JVM98," http://www.spec.org/osg/jvm98/.

[19] "SPECjAppServer2002," http://www.spec.org/jAppServer2002/

[20] "TPC-C Benchmark Specification," http://www.tpc.org/.

[21] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla, "Object- Oriented Architectural Support for a Java Processor," Proc. 12th European Conf. Object-Oriented Programming, pp. 430-455, July 1998.

[22] O. Waddell and R. K. Dybvig, "Fast and Effective Procedure Inlining", Proc. 1997 Static Analysis Symposium (SAS '97), Sept. 1997, pp. 35–52. Springer-Verlag Lecture Notes in Computer Science vol.1302.

[23] 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.

[24] "Pentium 4 Proc Optimization Guide," http://developer.intel.com/design/pentium4/manuals/