Loop Transformations in the Ahead-of-Time Optimization of Java Bytecode

Simon Hammond and David Lacey

University of Warwick

Abstract. Loop optimizations such as loop unrolling, unfolding and invariant code motion have long been used in a wide variety of compilers to improve the running time of applications. In this paper we present a series of experimental results detailing the effect these techniques have on the running time of Java applications following ahead of time optimization.

We also detail the optimization tools and transformations developed for this paper which extend the SOOT framework discussed in a number of previous papers on the subject.

Our experimentation, conducted on the SciMark 2.0 benchmarking suite, demonstrates that when optimized using the techniques mentioned, Java applications can benefit from performance improvements of up to 20%.

We finish with a discussion of the results obtained, including results on how the optimizations affect JIT compilation and class size and proceed to argue that ahead-of-time loop unrolling and unfolding optimization may have a role to play in improving the performance of Java applications, particularly in scientific applications.

1 Introduction

Improving the running time of programs through the optimizing phase of a compiler is a well established practice with a long history of well developed techniques and tools.

The success of optimizing compilers depends heavily upon the architecture that the final object code will be run on and architectures have changed since initial work in the field. One aspect of this phenomenon is that hardware architecture has become more complex but another aspect, and the one of interest here, is that in many cases the object code of a compiler is bytecode that is to be run on virtual machines such as the Java Virtual Machine.

Optimizing compiler designers face a couple of new issues when the target architecture is a virtual machine. Firstly, there is the question of whether the interpretive layer will affect the effectiveness of the optimizations. Secondly, when using a virtual machine we have the opportunity to compile and optimize at run-time and indeed this is what modern *just in time* (JIT) compiling virtual machines do.

The fact that compilation and optimization can happen at run time naturally leads to new thinking in compiler design. Do we need to perform ahead of time

A. Mycroft and A. Zeller (Eds.): CC 2006, LNCS 3923, pp. 109–123, 2006.

optimizations at all? One could argue that all optimizations can be deferred to run time and this would make the initial compiler design much simpler. One added advantage of this approach is that we have more information around at run time (*e.g.* profiling information) which can help us. However, it can also be argued that the static analysis required for some optimizations is too expensive to perform at run-time and should be performed beforehand. Another argument is that the interpretive layer slows down the program so much that optimizing the code is not worth it and that if one really wants speed then one should compile down to native code anyway - so called "way ahead of time" compilation.

All these possibilities have led to a lack of interest in traditional optimization of bytecode and despite much discussion about what the optimization architecture should be there is an underlying question to address. It is the authors' opinion that the discussion would be better informed if the following question was answered:

> Do traditional ahead-of-time compiler optimizations applied to bytecode cause significant performance increasing in running-time when executed on current (optimizing) JVMs?

i.e. given the current state of the art in JITs, does optimizing the bytecode still help? Is it still relevant? This is the question that this paper contributes towards answering. It builds on earlier work (particularly that of the SOOT optimization framework) and Section 4 details this earlier work that contribute toward answering this question. This paper provides a fully detailed study into a selection of traditional loop optimizations and the conclusion of the paper can be summarized as:

- Traditional ahead of time loop optimizations (in particular, loop unfolding and unrolling) applied to Java bytecode increase the run-time performance of the tested benchmark programs by up to 20%, with an average increase of approximately 4-5%

In the rest of this paper, Section 2 will give the methodology and decisions made when performing our tests. The numerical results of the testing will be given in Section 3 along with an analysis of these results. Finally, Sections 4 and 5 give the background to and a summary of the paper respectively.

2 Optimizations, Methodology and Testing

The research presented here is an empirical study of optimization. It is essential to such an endeavor that we identify the parameters of the experiments we carry out. In this case we can identify four main factors that affect the results:

- The benchmark suite used for testing.
- The optimizations used.
- The underling hardware architecture.
- The Java Virtual Machine (JVM) that executes the bytecode.

Each of the following sub-sections will describe the approach we took when considering each of these factors.

2.1 The Benchmarks

For the purposes of experimentation we used two benchmarking suites. The first was the command line version of the SciMark 2.0 benchmarking suite [19]. The second was the SPECjvm98 suite of benchmarks [3].

Table 1. Benchmark Description

Benchmark	**Description**
FFT	Performs a one-dimensional forward transform of 4000 complex numbers.
SOR	Performs Jacobi Successive Over-relaxation on a 100x100 grid
Monte Carlo	Approximates the value of Pi through integration of a quarter circle
Sparse Matrix Multiplication	Performs a matrix multiplication of a 1000x1000 square matrix containing 5000 non-zero elements
LU	Computes the LU Factorization of a dense 100x100 matrix using partial pivoting.
Compress	Lempel-Ziv Compressor/Decompressor
Jess	A Java Expert Shell
DB	An in memory database
JavaC	A JDK 1.0.2-complaint Java Compiler
MPEGAudio	Performs MPEG-3 Audio Compression
MTRT	A dual-threaded version ray tracing algorithm
Jack	A Java Parser Generation which has since become the JavaCC Project

Table 1 lists the benchmark suites optimized for this paper. We decided to choose a range of benchmarks across two benchmarking suites to ensure that the optimizations tested during our research provided benefits to a number of applications rather than a niche set of code. The SciMark 2.0 suite provides code mainly from scientific applications, these were chosen since they suited the type of optimization we were testing.

To obtain a average performance indicator for our benchmarks we averaged the benchmarks over 50 successive runs of the benchmarking suites.

Both benchmarks provide pre-compiled class files which have been compiled using the Sun Microsystems JDK 1.2 compiler.

2.2 Optimizations

We implemented three optimizations as intra-procedural transformation extensions to the SOOT Framework [21], a framework for analyzing and optimizing Java bytecode. This framework has an established and widely used set of tools for experimentation and research and as such was a natural choice.

All of the optimizations developed as part of our experiments were implemented as intra-procedural transformations in the Jimple intermediate representation [23] provided by the SOOT framework.

The three main optimizations implemented for this paper were:

- Loop Unrolling
- Loop Unfolding
- Loop Invariant Code Motion

The rest of this sub-section is dedicated to explaining each of these optimization techniques. Although brief descriptions are given, the techniques used are quite generic and more in depth explanations can be found in a number of books on the subject of compilation and optimization [1, 13, 16].

We choose to apply the loop invariant code motion transformation first as this removed redundant invariant statements prior to unfolding and unrolling. If invariant statements did exist they were hoisted before the unfolding and unrolling transformations to reduce the total size of the unfolded and unrolled copies of the loop.

Unfolding and unrolling are not commutative in that unfolding an unrolled loop leads to more copies of the loop being made. Although the unfolding factor can be altered to take this into account, we chose to apply the unfolding transformation before the unrolling process.

Loop Unrolling. Loop unrolling replaces the main body of a loop with several copies, adjusting the loop control code such that new body executes the same instructions as the original loop but with a smaller proportion of execution spent on evaluating the control. Since, by completing the unrolling process, we have changed the body of the loop to execute more than one original iteration on each iteration of the new loop we introduce a *epilogue* to the loop to handle 'odd' iterations which cannot be processed in the new unrolled loop [1]. The epilogue is created by placing a copy of the original loop at the end of the unrolled loop so that when execution is finished in the optimized code the epilogue can 'mop' up the remaining iterations.

Unrolling has two main benefits, firstly the transformation usually results in a smaller proportion of time being spent evaluating the control code of the loop since each iteration in the 'new' loop is executing the unrolling factor more iterations of the original. Secondly, the unrolling transformation opens multiple iterations to further optimization using techniques such as common sub-expression elimination [17]. Although common sub-expression elimination is not conducted by our optimization tool, the unrolled code may introduce opportunities for the just-in-time compiler to further optimize the code at runtime.

The level of benefit that the unrolling transformation provides is determined by the *unrolling factor* - i.e. the number of copies of the loop that replace the original body. For very small loops with simple control code a high unrolling factor can be used as the unrolled code is likely to fit into a cache line. For large complex loops small unrolling factors, usually 2, should be used as the unrolled loop can become too large to cache efficiently.

Original:

```
for(int i = 0; i < 10; i++) {
    a = a * 2;
}
```

Transformed:

```
for(int i = 0; i < 10; i = i + 2) {
    a = a * 2;
    a = a * 2;
}
```

Fig. 1. Unrolling applied to a simple loop

Due to the increase in the code size of a loop following unrolling the optimization can have detrimental effects on performance by making the code too large to fit into cache lines in the processor. When this occurs cache blocks may need to be transfered resulting in slower execution.

For our transformation we used a generic unrolling algorithm [16] to process single-basic block loops with simple control code. The unrolling factor used in our tool was set to 2 to limit the effect that larger loops would have on the caching of our benchmark programs.

The transformation was also developed to use a relatively conservative approach to unrolling in that it would only unroll loops without branching statements in their main body. This restriction was imposed because branching within loop bodies results in jump statements being fed through to the processor pipeline, as the unrolling process was meant to remove jump instructions by reducing the number of times the loop needed to jump to the start, allowing branches to be unrolled would be unlikely to result in a performance improvement.

We also chose a relatively simple approach to finding induction variables for the unrolling process. The tool was designed to only unroll loops with locally defined induction variables that are not compared to the result of a method return within the loop guard. These restrictions were created because the dynamic-class loading feature in Java allowed classes loaded at runtime to interfere with non local variables in a manner that could not be determined at optimization time. Since our transformation was intra-procedural we could not determine whether the return value of a method at runtime was constant therefore, induction variables compared against method returns could have behaved in a manner that would have meant the transformation resulting in different behaviour to the input.

Loop Unfolding. Loop unfolding, or *loop peeling*, removes a number of the first iterations of a loop and places them before the main body to form a *prologue*[18]. Extra control code is often introduced to ensure the overall number of iterations executed does not differ from the original loop code.

Original:

```
for(i = 0; i < 10; i++) {
  a = a * 2;
}
```

Transformed:

```
a = a * 2;
a = a * 2;
for(i=2; i < 10; i = i++) {
  a = a * 2;
}
```

Fig. 2. Unfolding applied to a simple loop

Unfolding has two main benefits, firstly it allows the earlier iterations of the loop to execute without requiring the processor to follow jump instructions back to the beginning of the loop, improving the ability of the code to be pipelined and secondly, opening the earlier iterations to further optimizations such as common sub-expression elimination.

In a similar manner to unrolling, unfolding can increase the overall size of the application code which can affect how the application will be cached. For large loops unfolding will create a sizable epilogue which may make the method difficult to cache. The impact of unfolding is usually determined by the *unfolding factor* - i.e. the number of copies of the loop placed in the prologue. Large unfolding factors can increase the size of the code considerably disrupting cache behaviour and may, in the case of Java, prevent the JIT compilation process if the JVM decides that the code is too large to compile on the fly.

We decided to use an unfolding factor of 8 to create a balance between unfolding enough iterations for the optimization to be useful yet keeping the factor small enough to prevent excessive increases in code size. Our unfolding transformation also used conservative approaches to deducing induction variables for the same reasons outlined in the unrolling optimization description. We did however allow the unfolding transformation to unfold loops with branches in the main body. The purpose of this decision was to permit optimizations to be carried out across the unfolded iterations with the possibility of reducing the number of branches through optimization on the branch conditions.

Loop Invariant Code Motion. Loop invariant code motion is applied to code within loops that does not change on each iteration of the loop [2], this is code whose execution is independent of the loop induction variable.

The transformation works by finding expressions using only constants or variables that are defined from outside the loop. Given that these values will not change on each iteration of the loop, any expression that uses only these values will also be unchanged by each iteration or change in a pre-determined manner. A generic algorithm is explained in [1].

When loop invariant code has been found it can be removed from the loop by hoisting. The hoisting process takes invariant statements and places them outside of the loop making adjustments to ensure the value assigned to any variables resulting from the invariant expression will be the same. For instance, if a variable increases by a value of 2 on each iteration of the loop then hoisting will set this variable to have a value of 2 multiplied by the number of iterations added immediately following the execution of the loop.

Original:

```
int a = 0;
int b = 0;
int c = 10;

for(i = 0; i < 10; i++) {
  a = a + 2;
  b = c * 2;
}
```

Transformed

```
int a = 0;
int b = 0;
int c = 10;

for(i = 2; i < 10; i = i++) {
  a = a + 2;
}
b = c * 2;
```

Fig. 3. Loop invariant code motion applied to a simple loop

Loop invariant code motion usually results in faster execution of loops because the redundant code is eliminated from being executed multiple times saving processor resources.

Our transformation used a conservative approach to finding invariant code by only searching for locally defined variables as potential candidates for invariance since any method calls from inside the loop could have changed field level values potentially resulting in an unsafe transformation.

2.3 The Hardware and Environment

We ran our benchmarks on two different architectures to check whether the results obtained would show similar trends and whether the underlying architecture of the processor would change the benefit the optimizations could bring.

Our first machine was a Pentium 4 2.4Ghz machine configured with 1Gb of RAM running Microsoft Windows XP Service Pack 2. The Pentium processor used in this machine contains two 16kb L1 caches one allocated to data entries and one to instruction entries.

Our second machine was an Apple G4 Powerbook equipped with a 1.5Ghz G4 Processor and 512Mb of RAM running Apple OSX Tiger. The G4 processor uses two 32kb L1 caches allocated to data and instruction.

2.4 The Java Virtual Machines

On the Windows XP Machine, we used the standard Sun Microsystems Java Standard Edition (J2SE) Version 1.5.0 without any extra configuration. On the Apple Powerbook machine we used the standard Apple 1.4.2 Virtual Machine.

Since we are interested in the relationship between the ahead of time optimizations and the optimizing compilation in the JVM, for the SciMark 2.0 Benchmarks we decided to experiment with both the Client and Server just in time compiler included with the standard virtual machine. The Client compiler is configured to carry out a smaller amount of class file analysis during startup in an effort to reduce the loading time of Java applications. Since less analysis is being carried out on the bytecode a smaller number of transformations can be conducted during execution.

The Server compiler takes an alternative strategy seeking to spend longer conducting analysis during startup with the assumption that the application is likely to execute for a longer period of time. As more analysis is carried out a larger number of transformations are available during execution.

The exact details of the compilers provided with the Sun and Apple Virtual Machines go far beyond the remit of this paper, we refer the reader to the respective vendor websites for up-to-date information and features.

3 Results

In this section we present our results and offer an analysis on the figures shown. Tables 2 and 3 show the performance change following our optimizations. The figures shown represent speedup which is computed as a factor of the time taken to compute the unoptimized code. A speedup for less than 1 indicates a performance degradation, a speedup of greater than 1 represents an improvement.

The aim of our experiments was to examine whether, through the use of traditional loop optimizations, the performance of Java applications could be improved. As the reader can see in the results tables, the effect of these optimizations is somewhat varied depending on the type of application being optimized.

The application that gained the most through optimization was the LU Factorization application included in the SciMark 2.0 suite. The results showed a 14% improvement on the Intel system and 20% improvement on the Apple G4 System. If we examine the source code of the LU application we notice a reasonably large number of loops which contain only a few instructions. Since the unrolling

Table 2. SciMark 2.0 Benchmark Result (Higher Result is Higher Performance)

	Pentium 4		G4	
SciMark 2.0 Benchmark	**Client JIT**	**Server JIT**	**Client JIT**	**Server JIT**
FFT	1.17x	1.12x	1.05x	1.04x
SOR	1.00x	1.03x	1.01x	1.01x
Monte Carlo	0.97x	0.99x	1.01x	1.02x
Sparse Matrix Multiplication	1.01x	1.09x	0.97x	0.97x
LU	1.14x	1.08x	1.20x	1.20x

Table 3. SPECjvm98 Benchmark Result (Higher Result is Higher Performance)

SPECjvm98 Benchmark	**Pentium 4**	**Apple G4**
Compress	0.99x	1.04x
Jess	1.10x	1.08x
DB	1.02x	1.03x
JavaC	1.06x	1.03x
MPEGAudio	1.00x	1.02x
MTRT	0.93x	0.95x
Jack	1.14x	1.11x

and unfolding of these loops produced a larger speedup than other benchmarks this indicates that the optimization of Java loops, like fully compiled languages, is best aimed at smaller, simple loops. Some of the benchmarks suffered from a performance degradation due to the transformations. The most notable of these was the MTRT ray-tracing benchmark from the SPECjvm98 benchmarks which suffered a 7% performance degradation on the Pentium 4 architecture.

Overall, the impact of the optimizations on many of the benchmarks was mixed but generally beneficial. Some benchmarks responded to optimization very well producing more efficient code that executed faster across a range of hardware platforms and virtual machines and other applications responded poorly. At the time of writing we have no firm conclusions about the nature of code that benefits or suffers from these transformations. However, the tool that applied the transformations did so indiscriminately on all loops that it could (given the innate conservative nature of the tool). A more guided transformation phase taking into account, for example, the size of the loop may lead to better results.

3.1 Architectural Considerations

The purpose of selecting an Intel Pentium 4 processor based machine and an Apple G4 processor based machine was to examine whether alternative virtual machines and architectures would alter the performance benefits of the optimizations presented.

As the reader can see from Table 2 the benefits vary slightly between the two machines. However, the overall average performance increase for the Pentium 4 and G4 architecture are roughly the same at 4.9% and 4.4% respectively.

Due to the fact that we are optimizing loops whose performance is largely down to efficient caching mechanisms we can attribute some of the variations in specific benchmarks to the different cache layouts offered by the two architectures.

Of course, hardware factors may be reduced by the effect of the bytecode running on an interpreter in the JVM. Nevertheless, there are two reasons why hardware factors may "show through": Firstly, in the case of caching, the fetching of bytecode instructions in a small loop will only take up a relatively small part of the data-cache leaving the rest for the program data and, secondly, we expect most of the critical code in these loops to be JIT compiled and therefore be run directly on the hardware anyway.

Another potential source for differences in the results comes from the varying number of registers available on the processors used for testing. The RISC based G4 architecture contains more registers than the Pentium and therefore may be able to hold more variables within the processor reducing the number of cache transfers required. This facility may lessen the impact of the optimizations on smaller loops as all the variables will be within registers thus reducing the speed improvement offered by more efficient usage of cache following unrolling.

3.2 The Client Versus Server Just-In-Time Compiler

For the SciMark 2.0 Benchmarks presented in Table 2 we decided to experiment with both the Client and Server just in time compiler included with the standard

virtual machine. As one might expect, on both architectures the benchmarks ran faster with the Server JIT than the Client JIT. Despite this there was little difference in the percentage speedups caused by the optimizations. On the G4 architecture the speedups were identical and on the Pentium 4 architecture the average were similar at 5.8% for the Client and 6.2% for the Server. These results indicate that the difference in underlying optimization architecture for current JITs do not affect the beneficial effects of the loop transformations.

3.3 Overhead of the SOOT Framework

The overhead of converting the Java bytecode input to the SOOT Jimple Intermediate Representation and back out to bytecode introduces some penalties due to the complexities of creating the representation and re-generating the input from this. In the original paper introducing SOOT [21] Vallee-Rai *et al.* claimed that this overhead was between 1% and 2% of program execution time.

Table 4. Effect on performance after being parsed by SOOT Framework but no optimizations applied

Benchmark	**Speedup**
FFT	0.96x
SOR	0.98x
Monte Carlo	1.00x
Sparse Matrix	0.99x
LU	1.00x

Table 4 shows the potential impact on performance of the benchmarking applications being input to the SOOT framework and emitted without any optimizations being applied. The output of the framework is likely to be different to the input bytecode due to the conversion of the input into an intermediate representation that does not have an exact one-to-one mapping between a virtual machine bytecode and an element of the representation.

Experiments conducted on our benchmarks by reading the bytecode into the optimization tool and emitting it without completing any transformation are shown in Table 4. We believe that our results are broadly representative of the data provided by Vallee-Rai *et al.*

3.4 Cost of Performing the Optimizations

The main aim of this paper is to evaluate the effectiveness of the optimizing transformations under consideration. As such, the development of the analysis and transformation tool was not undertaken with compilation performance in mind. However, for the sake of completeness, Table 5 shows the time and memory required for the optimization tool to process the benchmark classes on the same Pentium 4 machines that was used in to execute the benchmarks.

Table 5. Time and memory requirements for optimization

Benchmark	Memory Required (Mb)	Time Required (Seconds)
FFT	22.236	9.01
SOR	23.688	11.08
Monte Carlo	17.160	3.36
Sparse Matrix	19.904	5.14
LU	18.102	4.24

The figures shown in Table 5 are unrepresentative of the resources that the analysis and transformation would need in a industry standard developed compiler. As such, it is hard to judge whether the transformations could be performed at runtime in a JIT compiler. However, given the analysis required for the transformations this seem unlikely.

Instead, we would claim that in situations such as this, where the optimization process may be lengthy, ahead of time optimization, or possibly ahead of time analysis, could provide a mechanism for communicating information to the just in time compiler as a guide to which optimizations could be used in each section of code. A recent study on the use of inter-procedural side effect analysis [14] demonstrated that when code was analyzed ahead of time and the information communicated to the just in time compiler a performance improvement of up to 20% could be achieved.

3.5 Effect of Optimization on Bytecode Size

Due to the fact that unrolling and unfolding optimizations result in copies of the loop body being replicated either before or into the loop the size of the code is expected to increase. Table 6 shows the increase in code size of the SciMark 2.0 benchmarking suite when unrolling and unfolding optimizations are applied. Due to the use of the SOOT framework a small increase in code size is attributable to using the Jimple intermediate representation which does not provide an exact mapping between input bytecodes and the bytecodes emitted.

The transformation tool will unfold a loop first by a factor of 8 and then unroll with a factor of 2. Furthermore, the tool only acts conservatively and will not unroll loops nested within another. Given this, we can expect that every loop transformed will have its size increased by a factor of 10.

Table 6. Size of bytecode before and after optimization

Benchmark	No Optimizations	Optimized	Percent Increase
FFT	2718 bytes	3630 bytes	33.55%
SOR	580 bytes	1176 bytes	102.76%
Monte Carlo	547 bytes	622 bytes	13.71%
Sparse Matrix	510 bytes	668 bytes	30.98%
LU	2166 bytes	4482 bytes	106.92%

3.6 The Relationship Between AOT Optimization and JIT Compilation at Runtime

In this section we aim to investigate how the ahead of time optimization process changes the JIT compilation that occurs at runtime. Our work for this section centres on the SciMark 2.0 benchmarks. Table 7 shows that following optimization the number of bytes compiled by the Just-In-Time compiler at runtime for the Client JVM on the Pentium P4 architecture. It can be seen from the table that every benchmark has a rise in the amount of code compiled. However, this is to be expected due to the increase in bytecode size described in the previous section. It seems a reasonable assumption that given a bigger class file the JVM will compile more bytes of code.

The optimizations in this paper target loops in the program and these are the parts of the program one expects to be JIT compiled. If after the optimization the same loops are compiled then it may be reasonable to expect all the bytes added to the class file by unrolling and unfolding to be JIT compiled as well. In this case, we would expect the program size increase to be roughly the same as the increase in the number of extra bytes compiled at run-time. Table 8 shows that this is not the case. In fact there seems to be no correlation between the increase in code size and the increase in amount of JIT compilation.

The FFT, Monte Carlo and LU factorization benchmarks show behaviour where the JIT compilation increase is less than the increase in bytecode size. This is perhaps due to the fact that not all the code added by the transformations is processed by the JIT. Despite this, both FFT and LU factorization show large increases in running time performance.

The SOR and Sparse Matrix transformations show an even more unusual phenomenon where the JIT compiles more code after the transformation than the

Table 7. Number of Bytes Compiled by the Client Just-In-Time Compiler at Runtime

Benchmark	Bytes Compiled		Percentage Increase
	No optimization	optimized	
FFT	614 bytes	963 bytes	56.84%
SOR	148 bytes	884 bytes	497%
Monte Carlo	66 bytes	76 bytes	15%
Sparse Matrix	96 bytes	362 bytes	277%
LU	542 bytes	982 bytes	81%

Table 8. Comparison between size increase and JIT compilation increase

Benchmark	Program Size Increase	Extra Bytes Compiled
FFT	912 bytes	349 bytes
SOR	596 bytes	736 bytes
Monte Carlo	75 bytes	10 bytes
Sparse Matrix	158 bytes	266 bytes
LU	2316 bytes	440 bytes

transformation added. Somehow, the re-arrangement of code has caused the JIT to fire more often.

These results give us no clear correlation between the transformations and the effect they have on the amount of work the JIT will do at run time. It is possible that the results may be clearer if more fine grained information was known about the JIT (in particular, if it were known exactly which pieces of code were compiled or optimized). However, the tools available for the JVMs used in this experiment could not provide this information.

4 Background and Related Work

The work in this paper builds on other work that perform optimizations on Java bytecode. The main systems the authors are aware of are Briki [5] (although this is mainly a JIT compilation framework), Cream [6], Jax [20] and SOOT [21, 22]. The tool developed for this paper was built on top of SOOT. With the exception of Jax (whose main purpose was code compression) all of these systems report beneficial effects on performance when transforming Java bytecode ahead of time. As far as we are aware none of these systems include loop unrolling or unfolding in their transformation sets. However, forms of loop invariant code motion are applied by some tools (including the SOOT framework).

Other related systems are optimizing compilers that compile Java to native code [9, 12] and bytecode manipulation tools [7, 15].

Descriptions of all three optimizations can be found in standard compiler texts [1, 13, 16]. Loop unrolling and unfolding has been investigated on several architectures (for example [2, 8, 10, 11]) but as far as we are aware has not been investigated on platforms where a bytecode machine is used.

5 Conclusions and Future Work

The main contribution of this paper is the creation of a tool to perform certain optimizing transformations and a detailed investigation to determine whether traditional loop optimizations (in particular loop unrolling and unfolding) provide performance benefits on current JVMs when applied to bytecode.

The results average out with the transformations causing a 4-5% perfomance increase. Some benchmarks responded very well with up to a 20% increase and the worst performance degradation was a 7% decrease in performance. Overall the figures suggest that these AOT transformations are beneficial to the efficiency of Java programs.

In addition to the main results about performance increases, we can observe the following from the experiments:

- Although there are some variations in individual benchmarks, the trends of performance increase are the same across the hardware architectures tested
- Although there are some variations in individual benchmarks, the trends of performance increase are the same across the types of JIT tested

- The overhead of the SOOT framework is in line with earlier reported work.
- The effect of the transformations on the amount of code that is JIT compiled is unpredictable though increased performance can occur even with less than expected JIT compilation.

These results together lend support to the argument that, in some sense, the AOT optimization process is independent of the underlying run-time architecture including the JVM. However, individual cases can vary and the relationship between the transformations and JIT compilation is still not understood and could well still be an important factor in the effectiveness of the transformations. Further investigation into this relationship seems warranted.

As mentioned earlier in the paper, it may be possible for these transformations to be integrated into JIT compilation though the program analysis required may be too costly. A method of communicating ahead of time analysis to the JIT such as suggested in [14] may work in this context.

The tool developed for the purposes of this paper is quite conservative in its application of the transformations and also applies them uniformly without taking into account any context such as the size of the loop being transformed. It is possible that a more aggressive or more guided tool would produce more reliably beneficial results and this should be investigated further. Furthermore, the factors of unrolling and unfolding have been fixed for the reported experiments here, taking values that have worked well on other platforms. The performance effect of these parameters in Java is still to be fully investigated.

Often, the loop optimizations here are described as being successful due to their exploitation of cache behavior. Given that these transformations can be beneficial even when executed on a JVM, it may be the case that other loop transformations that affect cache behavior (*e.g.* loop tiling, strip mining, loop fusion etc.) would also benefit on Java code, particularly in scientific applications. A survey of these types of transformation can be found in [2].

References

1. Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge Press, 2002.
2. David F. Bacon, Graham Susan L., and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.
3. SPEC JVM98 Client Benchmarks. World Wide Web, `http://www.spec.org/osg/jvm98/`.
4. J M Bull, L A Smith, L Pottage, and R Freeman. Benchmarking java against c and fortran for scientific applications. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 97–1005. ACM Press, 2001.
5. M Cierniak and W Li. Just in time optimizations for high performance java programs. *Concurrency: Practice and Experience*, 1997.
6. L. R. Clausen. A java bytecode optimizer using side effect analysis. *Concurrency: Practuce and Experience*, 1997.
7. Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with joie. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167–178. USENIX Association, 1998.

8. Jack W Davidson and Sanjay Jinturkar. An agressive approach to loop unrolling. Technical report, University of Virginia, 2001.
9. Jeffrey Dean, Greg De Fouw, David Grove, Vassily Litvino, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proccedings OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 31, pages 83–100. ACM Sigplan, 1996.
10. J. J. Dongarra and A. R. Hinds. Unrolling loops in fortran. *Software Practice and Experience*, 9(3):219–226, 1979.
11. J.R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1987.
12. Robert Fitzgerald, Todd B Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for java. Technical report, Microsoft Research, 1998.
13. D. Grune, H. Bal, C. Jacobs, and K. Langendoen. *Modern Compiler Design*. John Wiley and Sons, 2000.
14. Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using inter-procedural side-effect information in JIT optimizations. In R. Bodik, editor, *Compiler Construction, 14th International Conference*, volume 3443 of *LNCS*, pages 287–304, Edinburgh, April 2005. Springer.
15. Han Bok Lee and Benjamin G. Zorn. A tool for instrumenting java bytecodes. In *The USENIX Symposium on Internet Technologies and Systems*, pages 73–82, 1997.
16. Steven S. Muchnick. *Advanced Compiler Design and Implementation*, pages 559–563. Morgan Kaufmann, 1997.
17. B.V.Mohan Kirshna Reddy. A work bench for loop transformation. Master's thesis, Indian Institute of Technology, Kanpur, May 2001.
18. Litong Song and Krishna Kavi. What can we gain by Unfolding Loops? *SIGPLAN Not.*, 39(2):26–33, 2004.
19. SciMark 2.0 Benchmarking Suite. World Wide Web, `http://math.nist.gov/scimark2/`.
20. Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for java. Technical Report RC 21451, IBM Research, 1999.
21. R. Vallee-Rai, C. Phong, G. Etienne, H. Laurie, L. Patrick, and S. Vijay. Soot - a Java bytecode optimization framework, 1999.
22. Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In *Computational Complexity*, pages 18–34, 2000.
23. Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations.