Journal of Universal Computer Science, vol. 12, no. 7 (2006), 788-810
submitted: 28/1/06, accepted: 6/4/06, appeared: 28/7/06 © J.UCS

Our Experiences with Optimizations in Sun's Java Just-In-Time Compilers

Anderson Faustino da Silva
(Federal University of Rio de Janeiro, Brazil
faustino@cos.ufrj.br)
Vitor Santos Costa
(Federal University of Rio de Janeiro, Brazil
vitor@cos.ufrj.br)

Abstract: Modern Java Compilers, such as Sun's HotSpot compilers, implement a number of optimizations, ranging from high-level program transformations to low-level architecure dependent operations such as instruction scheduling. In a Just-in-Time (JIT) environment, the impact of each optimization must be weighed against its cost in terms of total runtime. Towards better understanding the usefulness of individual optimizations, we study the main optimizations available on Sun HotSpot compilers for a wide range of scientific and non-scientific benchmarks, weighing their cost and benefits in total runtime. We chose the HotSpot technology because it is state of the art and its source code is available.

Key Words: Dynamic compilation, Just-in-Time compiler, compiler optimizations
Category: C.4, D.3.4

1 Introduction

Modern compilers perform a number of code optimizations in order to improve space and time performance of compiled programs [Muchnick 1997]. Optimizations allow programmers to write at a higher level of abstraction, hence expressing their intentions more clearly. They further allow compilers to take best advantage of the target hardware. Note that, contrary to what the term might imply, the optimization process rarely results in code that is "optimal" by some measure, only in hopefully improved code. In general, it is undecidable whether an optimization will improve the performance of a particular program.

Java applications are an important example of the benefits of optimizing compilers. The Java language is a very popular object-oriented programming language. However, the standard interpreted Java execution model has performance issues. Interpreting a Java method can be rather slow because each bytecode requires executing at least a template consisting of several machine instructions, hence limiting achievable performance [Romer et al 1996]. Best performance requires compiling the bytecodes to machine code. Just-In-Time (JIT) compilers go one step further, by doing the compilation at run-time and only for highly used methods. The price to pay is that the overall execution time now suffers from the compilation overhead of the JIT compiler. Understanding JIT compilation technology thus requires deciding on **(1)** which methods to compile; **(2)**; when to compile them; and, of course, **(3)** choosing which optimizations to apply.

JIT technology can work as well as standard compiler technology for applications with long-running times, such as scientific code [da Silva and Costa 2005]. A question is whether JIT can perform as well for non-scientific code, which is dominant in actual real-world Java programs. Often, these applications can be expected to have complex access patterns and/or relatively short execution times. Given the pressure on compilation time, it is particularly important to understand which optimizations are most valuable, and how to control them. This work addresses this problem.

We study the impact of some optimizations applied by Sun HotSpot JIT compilers to reduce the execution time in Java programs. We focus on the Sun HotSpot compilers beause they are state of the art, and because they are open source. We focus on a number of well-known optimizations. Inlining is a particularly important optimization for Object Oriented applications. It reduces overheads in method call, while increasing the opportunity for further optimizations. Value Numbering replaces fully redundant computations with copies. The Sun HotSpot compilers can detect value equivalence, thus the expressions need not be textually identical. Conditional Expression Elimination simultaneously removes dead code and eliminates expressions throughout a program. Range Check Elimination allows the compiler to safely remove checking in situations where it can determine that the index must fall within valid bounds, arguably an issue with Java. ADL-based Spill uses Architecture Description Language (ADL) [Shaw et al 1995] [Garlan et al 1994] supplied CISC instructions during register allocation. Coalescing removes register copies. Peepholing examines a few instructions at a time, towards reducing the number of instructions of replacing instructions. We also analyze the impact of using on-stack replacement, which allows moving from interpreted to compiled code during method execution.

Our methodology was to first evaluate the global performance of the system through performance, and then study the effect of each optimization by knocking it off. For this purpose, we used both high-level and cpu-level profiling to best understand program execution. We studied performance on both the Sun HotSpot Client compiler and the Sun HotSpot Server compiler, and for the Intel Pentium 4 architecture, with several scientific and non-scientific Java workloads.

This work show not only how some optimizations increase the execution performance of the Sun's Java Virtual Machine, but its contributions are:

– *To measure the impact of the some optimizations, in several Java workloads.* Several works about Java evaluate the system with only the SPECjvm98 benckmark [SPEC 2005]. However, it is interesting to undertand the performance with several (others) benchmarks.

- *To show that it is not enough to apply some optimizations in an determined application, without adjusting them at characteristics of the application.*
- *To show some problems in Sun's Java Virtual Machine.* This implies in,
- *To suggest new directions of research.*

The rest of the paper is organized as follows. Section 2 gives an overview of both Sun HotSpot Client Compiler and Sun HotSpot Server Compiler. Section 3 describes the optimizations analyzed in this paper and an brief description about on-stack replacement. Section 4 gives experimental results with statistics and performance data. Section 5 summarizes some related works. And the last section, section 6, presents our conclusions.

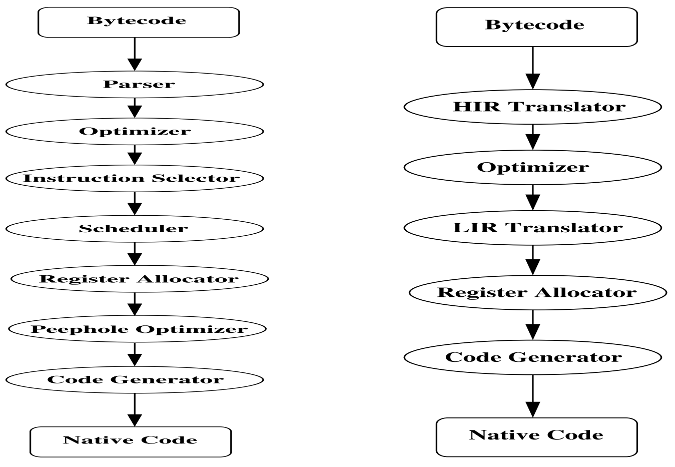
2 Just-In-Time Compilers

The HotSpot Virtual Machine (VM) [MicroSystems 2003] includes two different environments, the Client VM and the Server VM. The Java HotSpot Client VM is designed for running interactive applications and is tuned for fast application start-up and low memory footprint. The Java HotSpot Server VM is designed for maximum execution speed of server applications. Both share the same runtime, but include different just-in-time compilers, namely, the Client compiler and the Server compiler.

The Server compiler [Palecnz et al. 2001] is proposed for applications where the initial time needed can be neglected and only the execution time of the generated code is relevant. The Client compiler [MicroSystems 2003] achieves significantly higher compilation speed by omitting time-consuming optimizations. As a positive side effect, the internal structure of the Client compiler is much simpler than the Server compiler. Figure 1 depicts the structure of the Server compiler and Client compiler.

The structure of Client Compiler is separated into a machine-independent frontend and a partly machine-dependent backend. First, the frontend builds a high-level intermediate representation (HIR) by iterating over the bytecodes twice (similar to parsing process of the server compiler we discuss next). Only simple optimizations like constant folding are applied in this step. Next, the innermost loops are detected to facilitate the register allocation of the backend.

The backend converts the HIR to a low-level intermediate representation (LIR) similar to the final machine code. A simple heuristic is used for register allocation: it assumes that all local variables are initially located on the stack. Registers are allocated when they are needed for a computation and freed when the value is stored back to a local variable. If a register remains completely



(a) Server Compiler (b) Client Compiler

Figure 1: Structure of the Server compiler and Client compiler.

unused inside a loop or even in the entire method, then this register is used to cache the most frequently used local variable. This reduces the number of loads and stores to memory especially on architectures with many registers.

To determine the unused registers, the same code generator is run twice. In the first pass, code emission is disabled and only the allocation of registers is tracked. After any unused registers are assigned to local variables, the code generator is run again with code emission enabled to create the final machine code.

In contrast, the Server compiler uses an intermediate representation (IR) based on a static single assignment (SSA) graph [Appel 1998]. Operations are represented by nodes, the input operands are represented by edges to the nodes that produce the desired input values (data-flow edges). The control flow is also represented by explicit edges that need not necessarily match the data-flow edges. This allows optimizations of the data flow by exchanging the order of nodes without destroying the correct control flow.

The compiler proceeds through the following steps when it compiles a method: bytecode parsing, machine-independent optimizations, instruction selection, global code motion and scheduling, register allocation, peephole optimization and at last code generation.

The parser needs two iterations over the bytecodes. The first iteration identifies the boundaries of basic blocks. The second iteration visits all basic blocks and translates the bytecodes of the block to nodes of the IR. The state of the operand stack and local variables that would be maintained by the interpreter is simulated in the parser by pushing and popping nodes from and to a state array. Because the instruction nodes are also connected by control flow edges, the explicit structure of basic blocks is revealed. This allows a later reordering of instruction nodes.

Optimizations like constant folding and global value numbering [Gulwani and Necula 2004, Briggs et al. 1997] for sequential code sequences are performed immediately during parsing. Loops cannot be optimized completely during parsing because the loop end is not yet known when the loop header is parsed. Therefore, the above optimizations, extended with global optimizations like loop unrolling and branch elimination [Wedign 1984, Rastislav 1997], are re-executed after parsing until a fixed point is reached where no further optimizations are possible. This can require several passes over all blocks and is therefore time-consuming.

The translation of machine-independent instructions to the machine instructions of the target architecture is done by a bottom-up rewrite system [Pelegri and Graham 1988, Henry et al 1992]. This system uses the architecture description file that must be written for each platform. When the accurate costs of machine instructions are known, it is possible to select the optimal machine instructions.

Before register allocation takes place, the final order of the instructions must be computed. Instructions linked with control flow edges are grouped to basic blocks again. Each block has an associated execution frequency that is estimated by the loop depth and branch prediction. When the exact basic block of an instruction is not fixed by data and control flow dependencies, then it is placed in the block with the lowest execution frequency. Inside a basic block, the instructions are ordered by a local scheduler.

Global register allocation is performed by a graph coloring register allocator. First, the live ranges are gathered and conservatively coalesced, afterwards the nodes are colored. If the coloring fails, spill code is inserted and the algorithm is repeated. After a final peephole optimization, which optimizes processor specific code sequences, the executable machine code is generated. This step also creates additional meta data necessary for deoptimization, garbage collection and exception handling. Finally, the executable code is installed in the runtime system and is ready for execution.

3 Optimizations

The Sun HotSpot engines initialy interprete all methods and, based on runtime information collected during interpretation, identify the most frequently *hot* executed methods that deserve JIT compilation. This strategy is based on the observation that virtually all programs spend most of their time in a small range of code. Each method has a method-entry and a backward-branch counter. The *method-entry* counter is incremented at start of the method. The *backward* counter is incremented when a backward branch to the method is executed. If these counters exceed a certain threshold, the transition for a new version occur.

A number of optimizations can be performed by the engine. We focus only on a number of particularly important optimizations. Namely:

Inlining It [Waddell and Dybig 1997] is a well-kown technique that replaces calls to methods with copies of their bodies. It reduces the overhead of method invocation, which can be significant in object-oriented languages. It also enlarges compilation scope, exposing more optimization opportunities and eliminating the runtime overhead of creating a stack frame and passing arguments and return values. Excessive application of inlining can degrade performance. This is because of the increase in code size, which can severely decrease locality, hence decreasing the IPC (instructions per cycle). Explosive code growth can be avoided by concentrating only on hot spots. Heuristics or profiling feedback can further help in deciding which methods to inline. Experience has shown that choosing smaller methods is a good heuristic [Zhao and Amaral 2003]. If there is a single call site, inlining it should almost always result in reducing execution time. Inlining calls inside a loop is likely to provide significant opportunities for other optimizations. Constant-value parameters will also enable further optimizations on the inlined method.

Value Numbering Value numbering [Briggs et al. 1997] is one method for determining that two computations are equivalent and eliminating one of them. It associates a symbolic value with each computation without interpreting the operation performed, but in such a way that any two computations with the same symbolic value always compute the same value. Value numbering can operate on individual basic blocks or over the the entire method. The latter method, *global value numbering* uses SSA form [Cytron et al. 1991]. Muchnick [Muchnick 1997] presents an discuss about both local and global value numbering.

Conditional Expression Elimination Conditional expression elimination [Muchnick 1997] analyzes at compile time expressions and replaces the instructions for it results. It allows the compiler to eliminate constant-valued expressions and unreachable branches, resulting in dead code that can be later eliminated.

Range Check Elimination The Java language specification requires all array accesses to be checked at runtime, so that an attempt to use an out of range index will cause an exception to be thrown. The compiler can eliminate these checks [Gupta 1993] if it can prove that the index is always within the correct range, or if it can prove that an earlier check covers this check. If the compiler cannot prove this, the array reference must include range checking code. This ckecking can be very expensive for computation-intensive applications that heavily involve arrays, thus we would expect the optimization to be particularly useful for numerical applications.

ADL-based Spill Register allocator assigns the many temporaries to a small number of machine register. For this intention from an examination of the control and dataflow graph, an interference graph is derived. Now, it will be assigned a color to each node of this graph. It will be used as few colors as possible, but no pair of nodes connected by an edge may be assigned the same color. These colors corresponde to registers. If target machine has W registers, and it can W-colors the graph, then the coloring is a valid register assignment. But, if there is no W-coloring some variables and temporaries will be kept in memory instead of registers. When this occours is called spilling [Muchnick 1997]. Sun HotSpot Server compiler uses Architecture Description Language to assist the spill phase [Shaw et al 1995] [Garlan et al 1994].

Coalescing Register coalescing [Budimlic et al. 2002] is a well-known technique that eliminates copying betweeen registers. Coalescing searches the code for register copy instructions, such as: $R_1 \leftarrow R_2$. Upon finding such as instruction, it searches for the instructions that wrote to R_2 and modifies them to put their results in R_1 instead and removing the copy instruction.

Peepholing This technique inspects each instruction or sequence of adjacent instructions to determine if it may be replaced by a better instruction [Spinellis 1999]. The optimization is invoked by the machine-dependent modules, at a late stage.

On-Stack Replacement An on-stack replacement (OSR) [Fink 2003] mechanism enables a virtual machine to transfer execution between compiled versions, even while a method runs. Relying on this mechanism, the system can exploit powerful techniques to reduce compile time and code space, dynamically de-optimize code, and invalidate speculative optimizations. This techique is particularly important for long-running methods, such as the ones found in scientific code and in interpreters. The transitions from interpreted code to optimized code for long-running loops is based on invocations and loop counters. Additionally, the HotSpot compiler uses deferred compilation to avoid generating code for uncommon branches, such as the failed branch of class-hierarchy-based guarded inlining, and program points that invoke class initializers.

4 Experimental Results

In this section, we evaluate the impact of the optimizations we discussed in section 3. We used eighteen Java programs, nine of which (Crypt, FFT, LU, Heap Sort, Sparse, Euler, MolDyn, Monte Carlo and Raytracer) are part of Java Grande Forum Benchmark [JGF 2005]. The remaining nine (Antlr, Batik, Bloat, Chart, Fop, Hsqldb, Jython, Pmd and Xalan) are benchmarks in DaCapo Benchmark Suite [DaCapo 2005]. The table 1 presents an brief description of each applications and its problem size. All the measurements were obtained on an Intel(R) Pentium(R) 4 CPU 2.80 GHz uni-processor with 1024 MB memory, running Linux, and using the Sun HotSpot JVM version 1.5.0.

We also profiled application performance. We used both the Java profiler and use the CPU's performance counters in order to measure hardware instructions, memory reads and memory writes, cache misses, and instructions per cycles (IPC) of both the Client compiler and Server compiler. We used perfctr Intel/x86 hardware performance counters [PerfCTR 2005] for Linux with the associated kernel patch and libraries [PCL 2005].

Our first step was collect performance data for all optimizations we studied turned on and off. Next we investigated the individual optimizations. To do so, we knock off each individual optimization and measure its impact. After this, we analyzed the code size.

4.1 Client Compiler Versus Server Compiler

Figure 2 shows running times for these benchmarks, both with and without optimizations. We have four columns per application: the two left ones regard the Client Compiler, and the two to the right regard the Server compiler. The first column (leftmost) regards execution with all studied optimizations turned off, and the rightmost column gives performance with all optimization on. We further used the Java profiler to divide time among compilation time, garbage collection time, and execution time of compile and interpreted code.

Running times vary widely, between few seconds for Fop and for Euler to over a minute. The running time is most often dominated by the execution of compiled code. One exception is that applications such as Crypt, FFT, Sparse and Jython require on-stack replacement to take full advantage of the compiler. The second major exception is that we have long compilation times with Server compiler for some applications, such as Antlr (this also reflects in an increase in interpreted execution time).

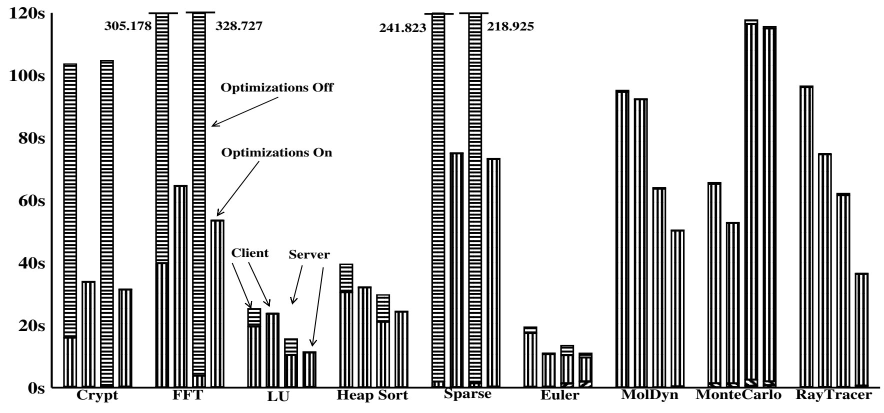
There is not a clear advantage between compilers. Regarding numerical applications, Crypt, FFT, Euler, and Sparse show similar performance. The Server compiler does better than Client compiler on Crypt, FFT, LU, HeapSort, Sparse, Euler, MolDyn and RayTracer. Surprisingly, it does worse than Client compiler

Benchmark	**Description**	**Problem Size**
Crypt	IDEA (International Data Encryption Algorithm) encrypton and descryption	50.000.000 bytes
FFT	One-dimensional forward transform	16M complex numbers
LU	Linear system factorization	2000X2000 system
Heap Sort	Heap sort algorithm	25000000 integers
Sparse	Sparse matrix multiplication	500000X500000 sparse matrix
Euler	Computational Fluid Dynamics	96X384 meshes
MolDyn	Molecular Dynamics simulation	8788 particles
MonteCarlo	Monte Carlo simulation	60000 sample time series
RayTracer	3D Ray Tracer	500X500 pixels
Antlr	Parses grammar files and generates a parser and lexical analyzer for each	320 grammars
Batik	Renders a number of SVG files.	6 svg files
Bloat	Performs a number of optimizations and analysis on Java bytecode files	82 Java files
Chart	Plot a number of complex line graphs and renders each one as PDF	52 graphs
Fop	Takes an XSL-FO file, parses it and , formats it generating a PDF file	1 XSL-FO file
Hsqldb	Executes a JDBC-like in-memory benchmark	20 clients 5 transactions per client
Jython	Interprets a series of Python programs	4 python programs
Pmd	Analyzes a set of Java classes for a range of source code problems	17 source codes
Xalan	Transforms XML documents into HTML	1 XML file

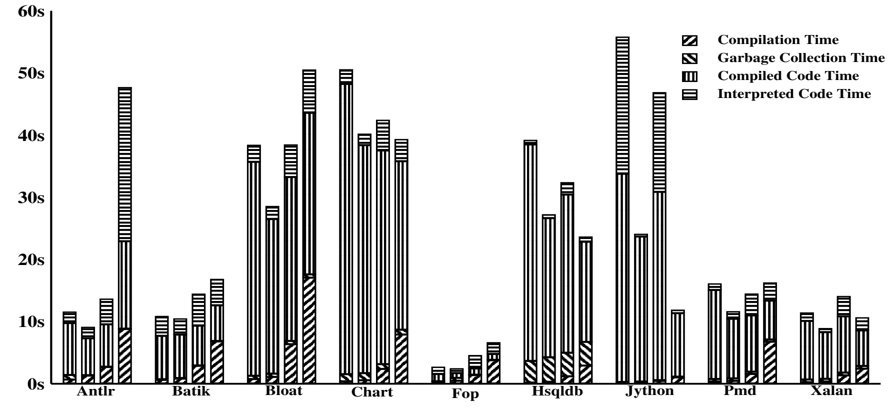
Table 1: Applications used in this paper.

on MonteCarlo, due to the number of cache misses. Compilation costs are always negligible for Client compiler on these benchmarks, and negligible for Server compiler except for the smallest application, Euler, which has the shortest running time and longest compilation time.

The optimizations we study have a significant impact in performance. We observed that Server compiler most often performs better than Client compiler with all our optimizations out. Server compiler includes a range of optimizations which we do not discuss in this paper. The exception is the applications that require on-stack-replacement to actually take advantage of the compiler.



(a) Scientific Applications



(b) Non-Scientific Applications

Figure 2: Execution time breakdown of both compilers.

The results for Java Grande applications do not carry to the DaCapo benchmarks. First, we can observe that Server compiler tends not to perform much better than Client compiler, even for applications with long running times, such as Bloat. This is due to 3 reasons:

1. Much longer compilation times: over 20 seconds for Bloat, and even higher for Antlr (some of the compilation time seems to be hidden in emulator time) . In general, compilation times are much longer for the DaCapo applications than for the Java Grande datasets, and much longer for Server compiler than

for Client compiler.

2. These benchmarks perform I/O and rely on system libraries, and thus, may spend a significant amount of time in native methods. A typical example is a method to deflate a ZIP archive, which takes about a quarter of total execution time in Chart.

3. In smaller benchmarks, a significant amout of time may be spent running interpreted methods for Server compiler, as Server compiler takes longer to compile a method than Client compiler. This happens with Fop, which spends about 10% of total time running a Java library method. This problem should not be an issue on longer applications.

Garbage collection time is most often negligible, except for Hsqldb, where it is about the same for the diverse versions. This benchmark is one example where Server compiler does well. The other example where Server compiler performs really well is Jython. Server compiler performs badly for Antlr, Batik, Bloat, Fop, Pmd, and Xalan, though.

Again, the optimizations matter. They result in improved performance for Client compiler over almost every application. On the other hand, the results for Server compiler are mixed. We see a substantial improvement on Jython, and a significant slowdown on Antlr.

In a nutshell, application runtime is most often better for Server compiler (except for Antlr, Bloat, Batik, and Fop), but compilation time is an issue for Server compiler in these applications, and can leave to major slowdowns.

Table 2 shows a low-level comparison between Server compiler and Client compiler. We show in both cases the number of hardware instructions executed, read and write memory accesses, number of cache misses, and IPC. It is interesting to compare the number of instructions executed and the IPC on both compilers. Server compiler usually executes less instructions and has a similar IPC for the majority applications, with best results for LU and Jython. There are a few surprises: Sparse runs less instructions in Client compiler. On the other hand, Sparse also has worse IPC for Client compiler, resulting in similar performance. MonteCarlo has the strangest result: Server compiler actually runs less instructions, but has a much worse IPC, partly caused by cache misses. The DaCapo benchmarks show the cost of extra compilation: Antlr and Fop show substantial increases in number of instructions executed.

Server compiler is usually very good at reducing the total number of read memory accesses. LU is very impressive: number of reads goes down by a factor of five. This also holds true for memory writes, although usually less so. Server compiler does perform really well on FFT, MolDyn, RayTracer, and Jython. Again, Sparse is the exception. Server compiler does perform significantly less

Benchmark	**Client Compiler**					**Server Compiler**				
	Inst (10^9)	Memory		Cache Miss (10^9)	IPC	Inst (10^9)	Memory		Cache Miss (10^9)	IPC
		Read (10^6)	Write (10^9)				Read (10^6)	Write (10^9)		
Crypt	81.29	24.16	28.36	4.91	0.86	69.82	22.10	25.04	6.39	0.80
FFT	21.86	6.67	2.68	1465.68	0.24	11.17	3.89	1.14	1482.04	0.15
LU	62.14	32.13	2.72	1536.04	0.85	13.63	5.44	2.70	3988.50	0.40
Heap Sort	35.39	16.88	3.15	122.89	0.89	19.98	4.89	1.64	148.21	0.87
Sparse	16.50	9.00	0.50	2148.55	0.08	19.50	10.50	1.50	2154.82	0.09
Euler	20.91	9.62	2.41	388.77	0.67	14.86	7.19	2.12	498.74	0.62
MolDyn	204.26	55.78	9.70	5796.12	0.79	109.39	32.59	2.23	4721.97	0.78
MonteCarlo	57.84	21.79	15.45	234.77	0.42	50.93	18.43	11.82	427.26	0.16
RayTracer	151.20	58.88	23.75	438.66	0.72	89.49	31.91	10.39	197.80	0.91
Antlr	16.23	5.96	2.64	92.62	0.95	24.14	3.28	1.53	87.27	1.92
Batik	17.34	9.06	2.39	239.74	0.89	18.23	8.74	2.30	228.02	0.95
Bloat	35.29	11.48	7.47	492.87	0.55	29.26	8.19	5.41	445.37	0.53
Chart	78.95	37.24	14.98	993.51	0.77	60.43	26.83	9.99	947.45	0.76
Fop	2.68	0.53	0.33	19.67	1.24	3.21	0.39	0.19	20.60	1.67
Hsqldb	46.15	17.92	8.48	397.19	0.97	30.00	9.11	3.85	252.11	1.07
Jython	73.79	21.28	12.43	210.26	1.14	29.68	8.70	4.75	172.94	1.09
Pmd	18.79	6.69	3.96	270.18	0.69	16.34	4.72	2.32	269.19	0.71
Xalan	17.47	7.46	2.95	196.61	0.87	15.74	5.53	2.49	164.43	0.94

Table 2: Behavior of the hardware.

memory accesses at Fop and Antlr, although it executes more instructions than Client compiler.

Both compilers inherit the same data structures from the interpreter, so they should have similar cache performance. This is mostly the case for the DaCapo benchmarks, but Server compiler often generates more cache misses in the Java Grande applications. Good examples are LU and MonteCarlo. This partly explains why both benchmarks have a much worse IPC in Server compiler. There are also examples where Server compiler is effective at reducing cache misses. Server compiler does very well on RayTracer and on Jython.

4.2 Impact of the some Optimizations

In this section we wanted to show only the difference in application time, not total running time. In the next figures, the leftmost column for each application refers to Client compiler, and the rightmost to Server compiler.

4.2.1 Inlining

Inlining is the most important optimization we study. Figure 3 shows the impact of using inlining switched on. Inlining is very effective for Client compiler, with up to 35% speedups in Hsqldb. Inlining works best for the DaCapo benchmarks, as they are written in a more object-oriented style: almost every application gains over 15%. The Java Grande applications benefit less. Only MonteCarlo, RayTracer, and to a lesser extent Euler have major speedups.

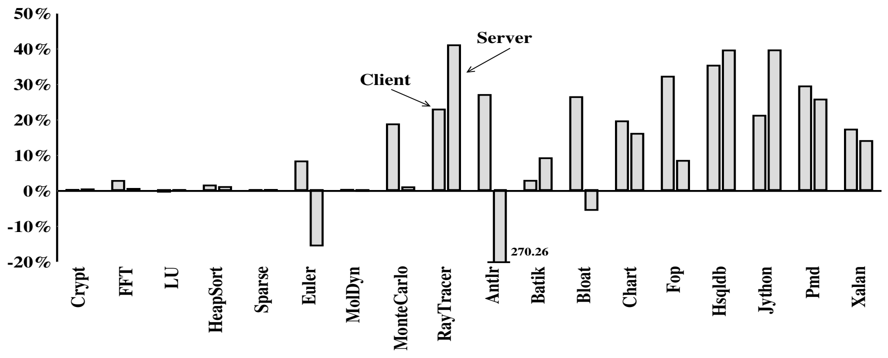


Figure 3: Impact of Inlining on Application Time.

Figure 3 shows a more complex picture for Server compiler. Whereas every DaCapo application benefited in Client compiler, Antlr and Bloat now have a slowdown. On the other hand, Jython benefits 40%, up from 20% in Client compiler. The Java Grande applications tell a similar story. Euler now has a significant slowdown, and MonteCarlo shows no benefit. Only RayTracer benefits from inlining.

We believe that the problem is that Server compiler can be too aggressive at performing inlining. This causes two problems. First, an increase in compilation time, which is also counted in. Second, excessive inlining may increase memory footprint and result in worse performance. A low-level analysis for Euler and MonteCarlo indicates this to be indeed the case (as indicated in table 2).

Inlining in Server compiler is controlled by several parameters. We experimented with tuning the variable *MaxInlineSize*, which gives the maximum size for inlining a method. The default value on our version is 35. We experimented with reducing this parameter, down to 5. This had a major impact on several applications. Namely, on Euler we obtained a speedup of 15% , for Antlr 21%, and for Bloat 13%. Unfortunately, this adjustment can worsen performance for other applications, such as Batik, where performance decreased from 9% to 2%. This suggests that we would like to tune the amount of inlining to the characteristics of individual applications.

4.2.2 Value Numbering

The impact of using value numbering with Client compiler is shown in Figure 4. Value numbering works best for numerical applications: Euler, Fft And Heapsort. It also works well for Fop. This optimization can improve the performance up to around 30% in Euler. Euler indeed seems particularly suited for this style of optimizations: it performs complex computations reusing the same value at several points. Non-numerical applications tend to benefit much less from value numbering. Note also that value numbering tends to reduce code size, so it interacts well with inlining.

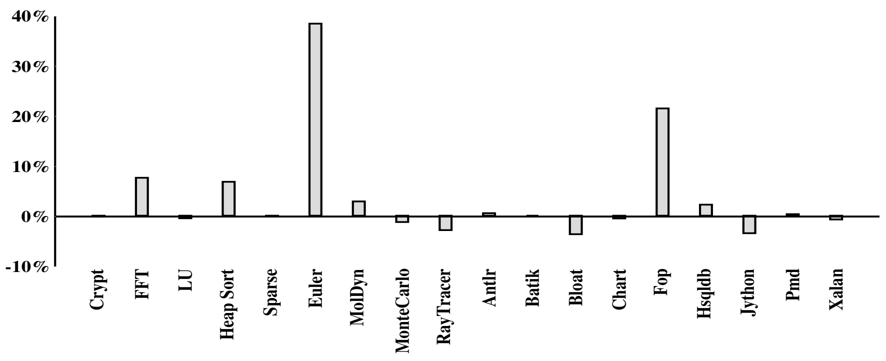


Figure 4: The impact of Value Numbering on Application Time.

4.3 Conditional Expression Elimination

We only measured conditional expression elimination on the Client compiler. The impact is shown in figure 5. Conditional expression elimination performs impressively well for Fop, where it achieves an over 20% speedup. It also benefits Xalan and Euler, but by much less. It is also interesting to remark that Hsqldb and Jython have somewhat performance. Otherwise, the optimization has a small effects on performance. We also found that quite often conditional expression elimination decreases the number of cache misses (probably by reducing code size).

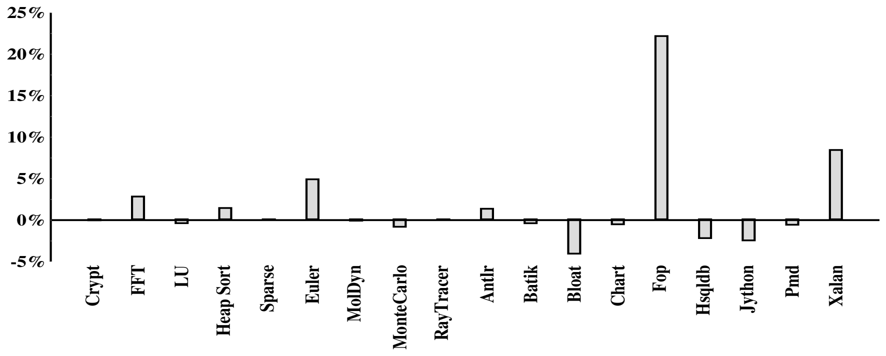


Figure 5: The impact of Conditional Expression Elimination on Application Time.

4.3.1 Range Check Elimination

Figure 6 shows the impact of range ckeck elimination. RCE has little impact on most benchmarks. Exceptions are Fop and FFT for Client compiler, and LU and Batik for Server compiler. Both FFT and LU can benefit from RCE. The Client compiler does well for FFT but not for LU, and Server compiler the other way round. This suggests that both algorithms can be improved.

The range check elimination can cause very significant slowdowns on Server compiler, namely for Euler. This seems to be caused by an interaction with inlining: both inlining and RCE increase code size, the combination of two seems to have even worse performance, both due to increased compilation time and to worse locality.

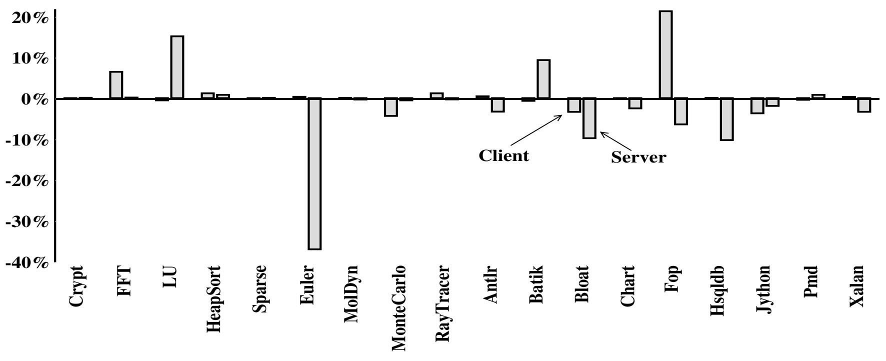


Figure 6: Impact of Range Check Elimination on Application Time.

4.3.2 ADL-based Spill

ADL-based Spilling is an server-only optimization. The impact of using ADL-based spill is shown in figure 7. Spilling is controlled by the ADL, Architecture Design Language, in this case written for a CISC platform. ADL-based spilling improves the performance up to 25%. In many cases the performance improvement is due to reducing the number of instructions executed. Antlr is a excellent case. This optimization, decreases the number of instructions executed in 15%, and also decreases both the memory accesses and cache misses. Spilling can have the usual problem of increasing compilation time, reducing overall performance.

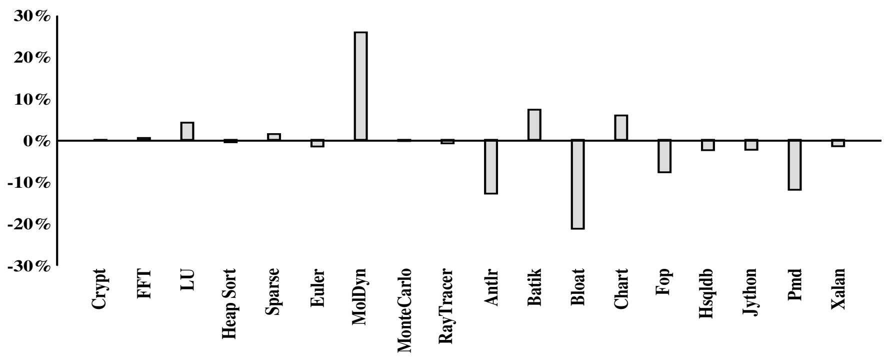


Figure 7: The impact of Spilling on Application Time.

4.3.3 Coalescing

Register coalescing is one server-only optimization. Its impact is shown in figure 8. Four applications benefit from conservative copy coalescing in the register allocator, with improvements ranging from 1.11% to 6.28%. In the cases, the optimization decreases the number of instructions executed and the number of cache misses. Unfortunately, coalescing can also work badly, again for the applications where code size and compilation time are a problem.

4.3.4 Peepholing

Server compiler and Client compiler implement very different forms of peepholing. In Client compiler peepholing works best for Fop and FFT, up to a factor of 6%. On the other hand, peepholing works in different ways for different applications. We found out that peepholing decreases IPC for Fop, while in improves

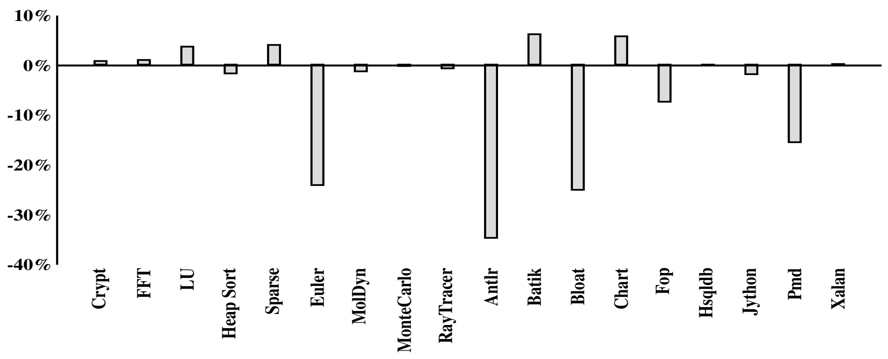


Figure 8: The impact of Coalescing on Application Time.

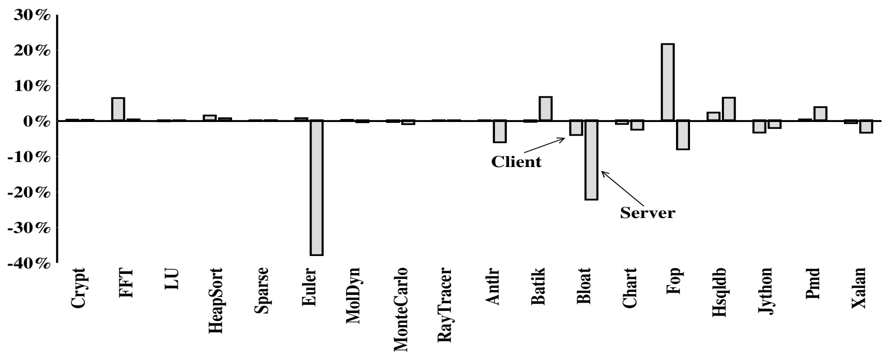


Figure 9: The impact of Peepholing on Application Time.

IPC on FFT. In both cases peepholing tends to increase the number of cache misses, even if it decreases the number of hardware instructions executed.

The results of peepholing for Server compiler are not very impressive. Again, the problem seems to be the interaction between peepholing and inlining. Both optimizations result in more code, so their effect is compounded when they are run together.

4.3.5 On-Stack Replacement

On-stack replacement is a JIT specific optimization, and as the results show, the second most effective optimization, with close up to 80% improvements on both compilers. Figure 10 shows the impact of on-stack replacement.

On-stack replacement is critical if the application spends most of the time in a single method, as it should have beeen made clear by Figure 2. This is indeed the case for applications such as Crypt, FFT, Sparse and Jython. It requires that more variables be kept a live, thus this can increase register pressure and pontentially decrease performance. This is not indeed the case for Client compiler, but for Server compiler, where Bloat and Pmd have worse performance. In these cases, the worse performance corresponds to a significant increase in compilation time.

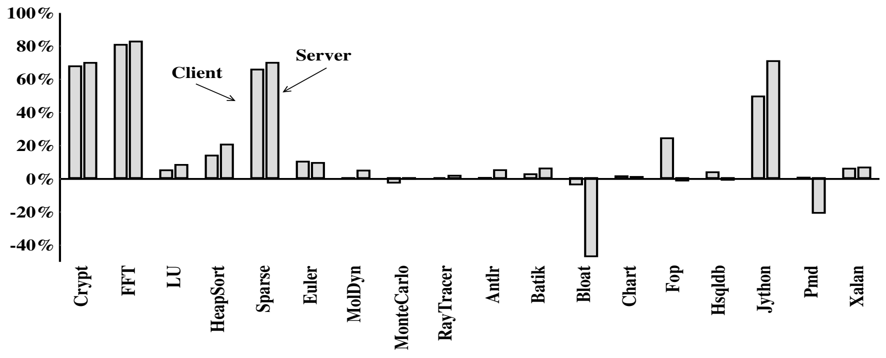


Figure 10: The impact of On-Stack Replacement on Application Time.

4.4 The Impact on the Code Size

We next discuss the variation in code size for the DaCapo applications. We do not discuss the variation in code size for the Java Grande applications, because code size tends to be stable.

Table 3 shows the variation in the code size in Client compiler. Applying on-stack replacement results in a very significant impact on code size, up to almost 30% in Xalan, which does not seem to benefit much from the optimization. Range check elimination substantially increases code size for Pmd, even though it does not affect performance much. Otherwise, it has a limited impact on code size. Last, inlining actually reduces code size in Client compiler. This is because inlining is not applied extensively, so the benefit from eliminating the instructions for frame manipulation dominates. The other applications have negligible effect on code size.

The table 4 confirms the results we discussed previously. The variation is much wider than for Client compiler. In the case of on-stack replacement the code can grow in up to 46% (Fop) and in the case of inlining the code size can more than double (Antlr). Notice that range check elimination now results in significant code compacting. Peephole compacts the code in up to 5%, coalescing up 3% and spilling in 4%. The exception is Antlr, where both coalescing and spilling increase code size.

	Inline	Value Numbering	Conditional Expression Elimination	Range Check Elimination	Peephole	On-Stack Replacement
Antlr	-1.66	-0.76	-0.04	0.91	-0.15	1.69
Batik	-2.17	0	0	0	-0.02	5.39
Bloat	-3.02	0.08	0.08	0.08	0.08	0.20
Chart	-2.90	-1.81	-1.81	0.02	0.01	7.20
Fop	-3.48	0	0	0	0	12.01
Hsqldb	-0.75	-0.08	0.07	0	0	14.41
Jython	-0.60	0	0.63	0	0	4.66
Pmd	-2.66	0	0	9.71	0	0
Xalan	-2.23	0	0	0	0	29.80

Table 3: Percentage of growth of the Client compiler's code size.

	Inline	Range Check Elimination	CISC Spill	Coalesce	PeepHole	On-Stack Replacement
Antlr	121.50	-7.99	12.46	41.95	-5.46	0.62
Batik	13.10	-1.68	-1.91	-3.10	-0.92	28.42
Bloat	16.99	3.47	0.85	-3.02	-5.14	-5.39
Chart	3.02	2.51	2.19	2.35	2.20	14.07
Fop	12.28	-1.99	-0.54	-0.96	-1.14	46.71
Hsqldb	0.99	-4.42	-4.42	-2.08	-1.08	9.05
Jython	6.74	-3.06	-3.06	-2.78	-3.06	13.71
Pmd	-7.12	0.91	-4.40	-3.05	-0.03	47.32
Xalan	2.21	0.24	0.40	0.40	-0.28	79.12

Table 4: Percentage of growth of the Server compiler's code size.

5 Related Works

The work that presents the Sun HotSpot Server Compiler [Palecnz et al. 2001] presents improvement for individual optimizations in a different context. Their work evaluated inlining, memory alias analysis and global value numbering. They show results for two target platform, namely: SPARC and IA32. For SPECjvm98 [SPEC 2005], in IA32 plataform, the improvement varied from 1% to 20% for the majority of the applications, except one that it possess an improvement of 60% when applying inlining. The same optimizations when applied in SPARC platform for the same benchmark improve the performance from 8% to 40% in the majority of the cases. However, as in IA32 the performance can be improved up to 72%. Their work suggests that inlining is the optimization that causes more impact in performance, and that the optimizations are more effective in SPARC machines. Unfortunately, their work does not discuss in detail the impact of each optimization.

The IBM compiler [Suganuma 2000, Ishizaki et al. 1999] implements adaptive optimization in a fashion similar to the HotSpot compiler. This work presents improvement for individual optimizations, it evaluates the effectiveness of some optimizations such as exception check elimination, simple type inclusion test, common sub-expression elimination, inlining of static method call, and resolution of dynamic method call. Experimental results have been shown that this optimizations are very effective for several types of programs. For SPECjvm98, one individual optimization improves the performance from 8% to 20% depend on the application. The results shown that inlining of static method call and resolution of dynamic method call are the best optimizations, while that simple type inclusion has the least benefit. Overall, the IBM JIT compiler combined with IBM's enhanced Java Virtual Machine is widely regarded as one of the top performing Java execution environments.

JUDO [Cierniak et al. 2000] is a high-performance VM that features multiple JITs compilers, and implements a dynamic recompilation mechanism. JUDO differs from the IBM and Sun compilers because it does not emulate bytecodes, but they are compiled to native code. It evaluates the impact of checkcast optimization, bounds checking elimination and inlining in SPECjvm98. In JUDO checkcast and bounds checking possess a low impact. Checkcast optimization improve in 3.4% the performance for one application, and it has negligible impact for the rest. Bounds checking elimination has only impact of 3.0% for on application. Inlining is the best optimization, it improves the performance from 4.2% to 26.70%.

It is clearly in these three works that inlining is the optimization that cause more impact in the performance, and that the architecture of the execution environment and the implementation of the optimization limits the impact of the optimization.

Other works that also describe a JIT Java platform, but they only present a comparison of its work with other existing ones, without detailing the impact of individual optimizations. This is the case of the works: Jalapeno [Alpern 2000, Jikes RVM 205], CACAO [Krall 1997], and LaTTe [Yang 1999].

Other interesting work [Suganuma et al 2003] presents an analysis of the impact of techniques used for a JIT compiler, however in a different context. The point is that although it has been assumed that methods are the units for compilation, this may not always be the best decision. The authors point that we should eliminate from the compilation target those portions that are rarely or never executed, focusing our efforts only on non-rare portions. Based on this strategy, their work implements a region-based compilation approach, which we would like to study.

6 Conclusions

We study the impact of compiler optimizations in two state of art JIT compilers for the Java language. We used the Java Grande and DaCapo benchmark suites, as they survey a large and diverse number of real-life Java applications. Studying the performance of JIT compilers is hard: we have all the issues in traditional compilers, plus the issues of choosing the best methods to compile, and the optimizations that give the best performance improvement, with the least overhead. Fortunately, the two Sun JIT compilers give us the chance to compare two very different compilers that both start from the same virtual machine code.

Our study was applied directed: we were interested in seeing in how different applications can benefit from optimizations. As expected, even reasonably small applications can have very significant performance benefits. On the other hand, our study shows a number of problems. First, optimizations can significantly result in a slowdown. Second, they can increase code size and memory footprint, generating sub-optimal performance. Inlining is a major example.

This work suggests several directions of research. First, fine control of the JIT is required. A case in point is on-stack replacement. It can bring significant benefits, but can also slow down applications which run short methods. It should be easy to control the compiler to only apply on-stack-replacement for methods which have very long running time. Manual control could be allowed (the user will often know whether on-stack-replacement is useful), but automatic discovery would be ideal. A more complex but important example is inlining. Clearly, inlining can be very useful. However, inlining can hurt performance, even for applications with long running times. Our work suggests that research is needed on how to best tune performance of inlining in a JIT environment.

Another research is to use profiling for feedback-directed optimizations [Smith 2000, Agesen et al. 1995] for allowing the compiler to compile (and optimize) only the executed path based on information extracted from the runtime system. Dynamic compilation take advantage of type feedback, because with this technique the compiler can determine which parts of an application should be optimized at all.

Dynamic compilation and feedback-directed optimizations improve both better runtime performance and interactive behavior. Using these techniques, object-oriented programs can execute efficiently without special hardware support, given the right compiler technology. Dynamic compilation is often regarded as complicated and hard to implement, but once the underlying mechanisms are in place, new functionality based on dynamic compilation can be added relatively easily. For example, a runtime profiler system could be implemented with relatively little effort because the system already supported dynamic compilation.

Our ongoing research is dynamically setting compilation parameters according to profiling input. This allow us to change method compilation strategies as we go along, given full execution data. Ideally, it should be possible to re-recompile methods, as execution goes along.

References

[Agesen et al. 1995] Agesen, O., Holzle, U.:"Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages"; Proc. of the Conference on Object-Oriented, (1995), 91-107.

[Appel 1998] Appel, Andrew W.: "Modern Compiler Implemenation" in C"; Cambridge University Press, USA (1998).

[Alpern 2000] Alpern, B., Attanasio, C. R., Barton, J. J., et al.:"The Jalapeno Virtual Machine"; IBM Systems Journal, 39, 1(2000), 211-238.

[Briggs et al. 1997] Briggs, P., Cooper, K. D., Simpson, L. T.: "Value Numbering"; Software emdash Practical and Experience, 27, 6(1997), 701-724.

[Budimlic et al. 2002] Budimlic, Z., Cooper, K. D., et al: " Fast Copy Coalescing and Live-Range Identification"; Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, (2002), 25-32.

[Cierniak et al. 2000] Cierniak, M., Lueh, G. Y., Stichmoth, J. M.:"Practing JUDO: Java Under Dynamic Optimizations"; Proc. of the ACM Conference on Programming Language Design and Implementation, (2000), 13-26.

[Cytron et al. 1991] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K.: "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph"; ACM Transactions on Programming Languages and Systems, 13, 4(1991), 451-490.

[da Silva and Costa 2005] da Silva, A. F., Costa, V. S.: "An Experimental Evaluation of JAVA JIT Technology"; Proc. of the 9th Brazilian Symposium on Programming Language, 2005, 45-60.

[DaCapo 2005] The DaCapo Benchmark Suite. http://www-ali.cs.umass.edu/DaCapo/gcbm.html; accessed December 2, 2005.

[Fink 2003] Fink, S., Qian, F.: "Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement"; Proc. of the International Symposium on Code Generation and Optimization, 2003, 421-452.

[Garlan et al 1994] Garlan, D., Allen, R., Ockerbloom, J.: "Exploiting Style in Architectural Design Environments"; Proc. of the ACM SIGSOFT Symposium on the Foundations of Softaware Engineering, 1994, 175-188.

[Gulwani and Necula 2004] Gulwani, S., Necula, G. C.: "Global Value Numbering Using Random Interpretation"; Proc. of the ACM SIGPALN-SIGACT Symposium on Principles of Programming Languages, 2004, 342-352.

[Gupta 1993] Gupta, R.: "Optimizing Array Bound Checks Using Flow Analysis"; ACM Letters on Programming Languages, 2, 1(1993), 135-150.

[Henry et al 1992] Henry, R. R., Fraser, C. W., Proebsting, T. A.:"Burs-Fast Optimal Instruction Selection and Tree Parsing"; Proc. of the Conference on Programming Language Design and Implementation, 1992, 36-44.

[Ishizaki et al. 1999] Ishizaki, K., Kawahito, M., et al.: "Design, Implementation, and Evalutiation of Optimizations in a Just-in-time Compiler"; Proc. of the ACM Conference on Java Grande, 1999, 119-128.

[JGF 2005] Java Grande Forum Benchmark. http://www.epcc.ed.ac.uk/javagrande, accessed in December 2, 2005.

[Jikes RVM 205] Jikes RVM Homepage. hppt://www-124.ibm.com/developerworks/oss/jikessrvm, accessed in December 3, 2005.

[Krall 1997] Krall, A., Graft, R.: "CACAO - A 64 bit Java VM Just-in-Time Compiler"; Proc. of the ACM Workshop on Java for Science and Engineering Computation, 1997, 362-387.

[MicroSystems 2003] Sun MicroSystems: Technical Report, Sun Developer Network Community, (2003).

[Muchnick 1997] Muchnick, S. S.: "Advanced Compiler Design and Implementation"; Morgan Kaufmann, San Francisco, 1997.

[Palecnz et al. 2001] Paleczny, M., Vich, C., Click, C.: "The Java HotSpot Server Compiler"; Proc. of the Java Virtual Machine Research and Technology Sumposium, 2001, 1-12.

[PCL 2005] Program Counter Library. http://www.fz-juelich.de/zam/PCL, accessed in December 2, 205.

[Pelegri and Graham 1988] Pelegri, Llopart E., Graham, S. L.: "Optimal Code Generation for expression Trees: An Aplication BURS Theory"; Proc. of the Conference on Priciples of Programming Languages, 1998, 294-308.

[PerfCTR 2005] Hardware Performance Counter. http://user.it.uu.se/mikpe/linux/perfctr, accessed in December 2, 2005.

[Rastislav 1997] Rastislav, Bodik, R. G, Soffa, M. L.: "Interprocedural Conditional Branch Elimination"; Proc. of the Conference on Programming Languages Design and Implementation, 1997, 146-158.

[Romer et al 1996] Romer, T. H., Lee, D., et al: " The Structure and Performance of Interpreters"; Proc. of the International Conference on Architectural Support for Programming Languages and Operating System, 1996, 150-159.

[Shaw et al 1995] Shaw, M., DeLine, R., et al.: " Abstractions for Software Architecture and Tools to Support Them"; Software Engineering, 21, 4(1995), 314-335.

[Smith 2000] Smith, M.: "Overcoming the Challenges to Feedback-directed Optimization"; Proc. of the Workshop on Dynamic and Adaptive Compilation and Optimization, 2000, 1-11.

[SPEC 2005] Standard Performance Evaluation Corporation. http://www.spec.org/osg/jvm98, accessed in December 2, 2005.

[Spinellis 1999] Spinellis, D.: "Declarative Peephole Optimization Using String Pattern Matching"; Proc. of the ACM SIGPLAN Notices, 1999, 47-51.

[Suganuma 2000] Suganuma, T., Ogasaware, T., et al.: "Overview of the IBM Java Just-in-Time Compiler"; IBM Systems Journal, 39, 1(2000), 66-76.

[Suganuma et al 2003] Suganuma, T., Yasue, T., Nakatani, T.: "A Region-based Compilation for a Java Just-in-Time Compiler"; Proc. of the Programming Language Design and Implementation, 2003, 23-45.

[Waddell and Dybig 1997] Waddell, O., Dybig, R. K.: "Fast and Effective Procedure Inlining"; Proc. of the Static Analysis Symposium, 1997, 35-52.

[Wedign 1984] Wedign, R. G., Rose, M. A.: "The Reduction of Branch Instruction Execution Overhead using Structured Control Flow"; Proc. of the Internation Symposium on Computer Architecture, 1984, 119-125.

[Yang 1999] Yang, B. S., Moon, S. M., et al.: "LaTTe - A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation"; Proc. of the International Conference on Parallel Architecture and Compilation Technique, 1999, 362-387.

[Zhao and Amaral 2003] Zhao, P., Amaral, J. N.: "To Inline or Not to Inline? Enhanced Inlining Decisions"; Proc. of Workshop on Languages and Compilers for Parallel Computing, 2003, 23-45.