Journal of Universal Computer Science, vol. 11, no. 7 (2005), 1291-1309
submitted: 28/4/05, accepted: 9/6/05, appeared: 28/7/05 *© J.UCS*

An Experimental Evaluation of JAVA JIT Technology

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: Interpreted languages are widely used due to ease to use, portability, and safety. On the other hand, interpretation imposes a significance overhead. Just-in-Time (JIT) compilation is a popular approach to improving the runtime performance of languages such as Java. We compare the performance of a JIT compiler with a traditional compiler and with an emulator. We show that the compilation overhead from using JIT is negligible, and that the JIT compiler achieves better overall performance, suggesting the case for aggresive compilation in JIT compilers.
Key Words: Dynamic compilation, Just-in-Time compiler, compiler optimizations
Category: C.4, D.3.4

1 Introduction

Interpreted languages are widely used due to ease to use, portability, and safety. On the other hand, interpretation imposes a significance overhead. Just-in-Time (JIT) compilation [Plezbert et al. 1997, Krall 1998] is a popular approach towards improving the runtime performance of such languages. JIT systems convert on-the-fly source code sequences into an equivalent sequence of the native code, allowing for signifant performance improvements. Notice that there is a cost: program execution time now includes the compilation overhead, which is paid before-hand in conventional compilers. This argues that JIT compiler should fast and light-weight, as well as being able to generate high-quality native code.

The Java language is the most popular example of JIT technology in action. The Java programming language [Arnold et al. 2000] was developed by Sun Microsystems [Sun 2003] as a general-purpose, object-oriented, concurrent language. Java was designed as a portable language that runs on multiple host architectures and allows secure delivery of software components. Although the syntax is similar to `C++`, it omits the most complex and unsafe features of `C++`. Instead, many sophisticated concepts were added to simplify development and increase security.

The emerging of the World Wide Web contributed much to the success of Java. The integration of small Java programs into web pages enables the designers to use a full-blown programming language and to develop interactive

applications that are seamlessly integrated in the web browser. On the other hand, transferring executable code over an untrusted network like the Internet requires careful checks before execution to guarantee that no virulent code is executed on the client, as enforced by the Java specification. This is particularly true as Java is used on a wide variety of systems, including small embedded systems such as mobile phones and PDAs.

To guarantee portability and platform independence, Java applications are not distributed in native code for a specific hardware platform. Instead, Java environments take advantage of the concept of a Java virtual machine (JVM) for abstraction. Java source code is compiled to a compact binary representation called Java bytecodes which is interpreted or compiled, using a JIT, by the JVM. The application is stored in a well defined binary format, the class file format, containing the bytecodes together with a symbol table and other ancillary information. The Java virtual machine is defined independently from the Java programming language, only the class file format connects these parts.

In this work we investigate the performance of current Java Virtual Machines, by comparing it with what can be obtained using a static compiler. First, we investigate the performance cost of using a JVM interpreter. Second, we investigate whether the performance of virtual machine with JIT can match what one would achieve with a conventional static compiler. We show that modern JIT technology for Java can indeed achieve excellent performance.

The rest of the paper is organized as follow. Section 2 describes a structure of a JVM. Section 3 presents the principles of Just-in-Time compilation. Section 4 describes the structure of the JIT compilers's Sun. Section 5 describes the structur of the GNU `GCJ` compiler. Section 6 summarizes our measurements and results. A Section 7 describes some related works. And finally, last section concludes our paper.

2 The Java Virtual Machine

Virtual machines are a widely known concept to obtain platform independence and to conceal limitations of specific hardware architectures. In general, a virtual machine emulates an abstract computing architecture on a physically available hardware. Because virtual machines are just a piece of software, the restrictions of hardware development are not relevant. For example, it is possible to extend the core execution unit with high-level components, e.g. for memory management, thread handling and program verification. The instruction set of a virtual machine can therefore be on a higher level than the instruction set of a physical processor. This in turn leads to a small size of the compiled code, where a single-byte instruction can perform a quite complex action.

The Java virtual machine [Lindhom et al 1999, Vernners 1999] is a stack machine that executes bytecodes. It defines various runtime data areas that are used

for the execution of a program. While some data areas exist only once per virtual machine, others are created for each executed thread.

When a Java virtual machine is started, the global data structures are allocated and initialized. The *Heap* models the main memory of a JVM. All Java objects are allocated on the heap. While the allocation of an object is invoked by the executed program, the deallocation is never performed explicitly. Instead, objects that are no longer reachable by the program are automatically reclaimed by a *garbage collector*. Java programs cannot cause memory errors such as memory leaks or accesses to already freed objects.

A class or method must be loaded into the JVM before it can execute. Classes are represented as complex data structures where a few parts happen to be sequence of bytecodes, a constant pool that acts as an extended symbol table, and miscellaneous data structures. The bytecodes of the class are loaded to the method area, shared among all threads. The constants are loaded to the constant pool.

Starting of a new thread implies the creation of the per-thread data structures. Because threads are part of the Java specification, each JVM must be capable of executing multiple threads concurrently. Basic means for the synchronization of threads are part of the Java specification. Each thread has its own stack and a set of registers, including the program counter.

3 Principles of Just-in-Time Compilation

Interpreting a method is rather slow because each bytecode requires a template consisting of several machine instructions, hence limiting achievable performance [Romer et al 1996]. Best performance requires compiling the bytecodes to machine code that can be executed directly without the interpreter. If compilation takes place while the program is being executed, it is called *just-in-time compilation, or JIT* [Plezbert et al. 1997].

A JIT compiler can use two approaches to translate Java code into native code: it can translates a method at a time as it sees them; or, it can initialy interprete all methods and, based on runtime information collected during interpretation, identify the most frequently *hot* executed methods that deserve JIT compilation. Generaly, a virtual machine that uses the second approach is called a *hotspot machine*.

The first approach is straightforward, we discuss the second method in some more detail. 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 method is scheduled for compilation. It is expected that counters of frequently executed methods, called the *hot spots* of a program, will soon reach the threshold and the methods will be compiled without wasting much time interpreting them.

On the other hand, methods that are executed infrequently, e.g. only once at the startup of the application, never reach the threshold and are never compiled. This greatly reduces the number of methods to compile. Thus, the compiler can spend more time optimizing the machine code of the remaining methods. The argument thus is that using a mixture of interpreted and compiled code guarantees an optimal overall performance.

A further advantage of using counters is that it guarantees that every method will be interpreted before it is compiled. So, all classes that are used by the method are already loaded and methods that are called are known. Additionally, the interpreter collects runtime information such as the common runtime type of local variables. This information can be used by the compiler for sophisticated optimizations that would not be possible if the methods were compiled before their first execution.

More precisely, notice that some highly effective compiler optimizations are complicated by the semantics of the Java programming language. For example, most methods are virtual, and they cannot be inlined because the actually called target is not known statically: the semantics of a call can change as classes are loaded dynamically into the running program. Nevertheless, a JIT compiler does perform inlining of such methods optimistically. The price to pay is that a compiled method may be invalidated when a new class is loaded. In such rare cases, the method is compiled again without this optimization. A further problem arises if the invalidated method is currently being executed and therefore stack frames of this method are active. A solution to this case is to allow one to switch back from the compiled code to the interpreter. This transition is called *deoptimization.* The compiler must create meta data that allows the reconstruction of the interpreter state at certain points of the compiled code.

Deoptimization allows the compiler to perform aggressive optimizations that speed up the normal execution, but may lead to situations where the optimization was too optimistic and must therefore be undone. There are some additional cases where a compiled method is deoptimized, e.g. when an asynchronous exception is thrown. The compiled code does not need to handle such complicated, uncommon cases. In the first approach, in cases where deoptimization is necessary, the JIT compiler recompiles the method and dispatches it.

As we said, a method is compiled when the counters of the method exceed a certain threshold. Typically, the decision is made before the execution of the method starts because no special handling is needed in this case to switch from the interpreted to compiled code: Instead of the interpreter, the compiled code

is called. But this solution is not always sufficient. When an interpreted method executes a long running loop, then it is necessary to switch to compiled code while a method is running [Fink at al 2003]. In this case, a special version of the method is compiled.

When the compiler encounter situations that occur rarely, but are difficult to handle, the compilation of the method is stopped and the execution is continued in the interpreter.

4 The Sun JIT Compiler

We use the SUN HotSpot compilers in our experiments for two reasons: it is one of the best performing compilers available for Java, and their source code is publically available. The HotSpot VM [Sun 2003] is available in two versions: the client VM and the server VM. The Java HotSpot Client VM is best 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 long running 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 [Paleczny et al 2001] is proposed for long running applications where the initial time needed for can be neglected and only the execution time of the generated code is relevant. The client compiler [Sun 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. It is separated into a machine-independent frontend and a partly machine-dependent backend. The structure of the Server and Client are depicted in figures 2 and 1.

The client compiler works as follows. First, the frontend builds a high-level intermediate representation (HIR) by iterating over the bytecodes twice (similar to the parsing of the server compiler). Only simple optimizations like constant folding are applied. 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: at the beginning it assumes that all local variables are 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 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

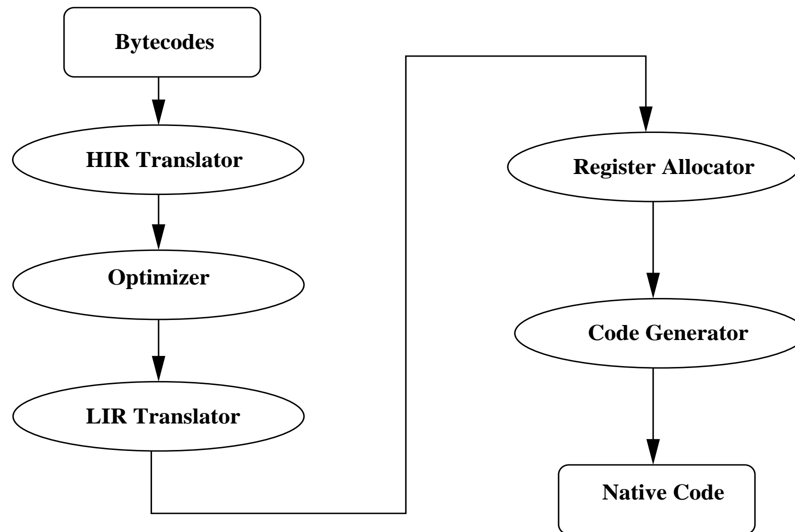

Figure 1: Structure of the Sun Client compiler.

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.

The server compiler [Paleczny et al 2001] is a fully optimizing compiler that performs all classic optimizations of traditional compilers, like common subexpression elimination, loop unrolling [Huang et al 1999] and graph coloring register allocation. It also features Java specific optimizations, such as inlining of virtual methods [White Paper 2004, Detlefs et al 1999], null-check elimination [Kawahito et al 2000] and range-check elimination. These optimizations reduce the overhead necessary for guaranteeing safe execution of Java code to a minimum. The compiler is highly portable and available for many platforms. All machine specific parts are factored out in a machine description file specifying all aspects of the target hardware.

The extensive optimizations lead to high code quality and therefore to a short execution time of the generated code. But the optimizations are very time-consuming during compilation, so the compilation speed is low compared with other just-in-time compilers. Therefore, the server compiler is the best choice for long running applications where the initial time needed for compilation can be neglected and only the execution time of the generated code is relevant.

The server compiler uses an intermediate representation (IR) based on a static single assignment (SSA) graph. Operations are represented by nodes, the

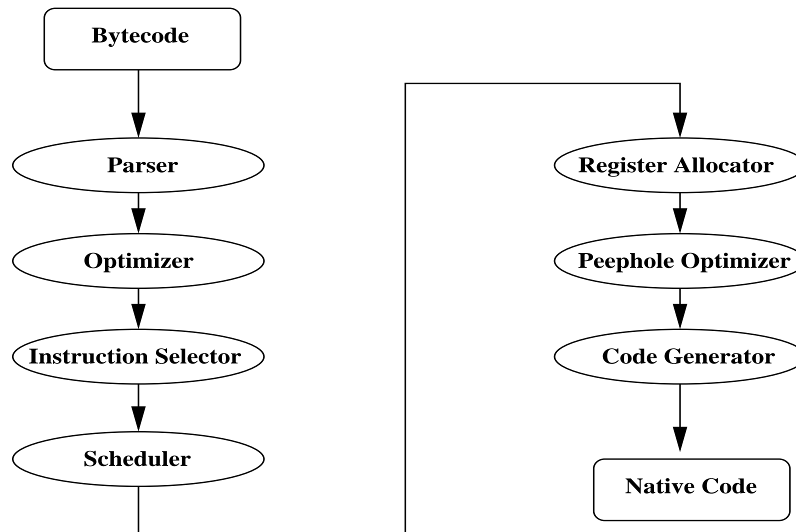

Figure 2: Structure of the Sun Server compiler.

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. A basic block is a straight-line sequence of bytecodes without any jumps or jump targets in the middle. 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 [Simpson 1994, Gulwani et al 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 et al 1984, Bodik et al 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-Llopart et al 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 processorspecific 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.

The server compiler provides an excellent peak performance for long running server application. However, it is not suitable for interactive client applications because the slow compilation leads to noticeable delays in the program execution. The peak performance is not apparent to the user because client applications spend most of their time waiting for user input.

5 GNU GCJ Compiler

The GNU Compiler Collection (GCC) [GCC 2005] is a set of compilers produced by the GNU Project. It is free software distributed by the Free Software Foundation. GCC is the standard compiler for the open source Unix-like operating systems. Originally named the GNU C compiler, because it only handled the C programming language, GCC was later extended to compile C++, Fortran, Ada, and Java.

The approach of the GCJ [GCJ 2005] Project is very traditional. GCC views Java as simply another programming language and compiles Java to native code using the GCC infrastructure. On the whole, compiling a Java program is actually much simpler than compiling a C++ program, because Java has no templates and no preprocessor. The type system, object model and exception-handling model are also simpler. In order to compile a Java program, the program basically is represented as an abstract syntax tree, using the same data structure GCC uses for all of its languages. For each Java construct, we use the same internal representation as the equivalent C++ would use, and GCC takes care of the rest. In this case, GCJ can then make use of all the optimizations and tools already built for the GNU tools.

The GCC compiler is divided into frontend and backend. The frontend parses the source code, produces an abstract syntax tree and applies optimizations. The backend converts the trees to GCC's Register Transfer Language (RTL), applies various optimizations, register allocation and code generation. The structure of the GCC (GCJ) is depicted in Figure 3.

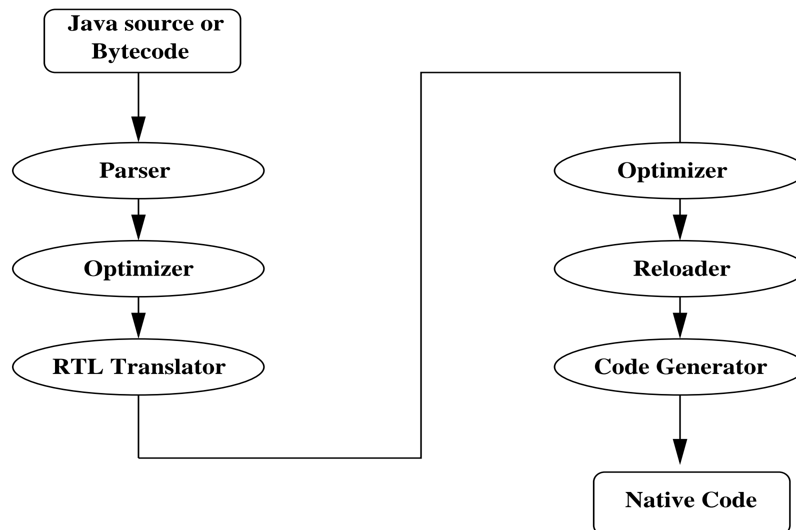

Figure 3: Structure of the GCJ compiler.

The frontend uses two forms of language-independent trees: GENERIC and GIMPLE [GIMPLE 2005]. Parsing is done by creating temporary language dependent trees, and converting them to GENERIC. The so-called gimplifier then lowers this more complex form into the simpler SSA-based GIMPLE form which

is the common language for a large number of new powerful language- and architecture-independent global optimizations. These optimizations include dead code elimination, partial redundancy elimination, global value numbering, sparse conditional constant propagation, and scalar replacement of aggregates.

The behavior of the `GCC` backend is partly specified by preprocessor macros and functions specific to a target architecture, for instance to define the endianness, word size, and calling conventions. The front part of the back end uses these to help decide RTL generation, so although `GCC`'s RTL is nominally processor-independent, the initial sequence of abstract instructions is already adapted to the target.

The exact set of `GCC` optimizations varies from release to release, but includes the standard algorithms, such as jump optimization, jump threading, common subexpression elimination and instruction scheduling. The RTL optimizations are of less importance with the recent addition of global SSA-based optimizations on GIMPLE trees, as RTL optimizations have a much more limited scope, and have less high-level information.

A "reloading" phase changes abstract registers into real machine registers, using data collected from the patterns describing the target's instruction set.

The final phase is somewhat anticlimactic, since the patterns to match were generally chosen during reloading, and so the assembly code is simply built by running substitutions of registers and addresses into the strings specifying the instructions.

6 Evaluation

We measured performance on an Intel Pentium 3 processor with 866 MHz, 512 KByte L2-Cache, 512 MByte of main memory, running RedHat Linux. In this evaluation we used Sun version 1.4.2 and GNU `GCJ` 3.4.1.

The Java Grande Forum benchmarks [JGF 2005] is a suite of benchmark tests designed towards measuring and comparing alternative Java execution environments. This suite uses large amounts of processing, I/O, network bandwidth, or memory. It includes not only applications in science and engineering but also, for example, corporate databases and financial simulations.

The benchmarks used in this paper, are chosen to be short codes containing the type of computation likely to be found in large applications and intending to be representative of large applications. This were suitably modified by removing any I/O and graphical components. Table 1 presents the suite applications and the problem size used in this paper.

To measure the performance of Sun JVM and GNU `GCJ`, we collect some informations during execution time, namely, cache misses, number of stalls, number of instruction, percentage of load-store instructions and instruction per cicle

Aplication	Description	Problem Size
Crypt	IDEA encryption and decryption	50.000.000 bytes
FFT	One-dimensional forward transform	16M complex numbers
LU	Linear system factorisation	2000X2000 system
SOR	Successive over-relaxation	2000X2000 grid
Euler	Computational Fluid Dynamics	96X384 meshs
MolDyn	Molecular Dynamics simulation	8788 particles
MonteCarlo	Monte Carlo simulation	60000 sample time series
RayTracer	3D Ray Tracer	500X500 pixels

Table 1: Benchmarks Data Size.

(IPC). We used the Program Counter Library (PCL) [Berrendorf et al 2003] to collect this data. The library relies hardware informations during collected program execution time.

To investigate the performance cost of using a JVM interpreter, we first compare the Sun JVM in interpreted mode with GNU `GCJ` compiler. We would expect the execution of compiled code to be faster than interpreted code. To obtain a fairer comparison, we turn off all optimizations in `GCJ`.

And, to investigate whether the performance of virtual machine with JIT can match what one would achieve with a conventional static compiler, we will realize some experiments comparing the Sun JIT JVM with `GCJ`. In this case, the `GCJ` compiler was used with all optimizations turned on, in other words, with the option -O2 turned on.

6.1 Sun Interpretation Versus `GCJ` -O0 Compilation

As we expected, compiled code is faster than interpreted code. The Sun Interpreter is between 3 to 11.6 times slower than `GCJ`, as shown in 4. The smallest difference is for the Euler application: compiled mode is only 3 times faster than interpreted mode. RayTracer has the major difference: 11.6 times faster than in interpreted mode.

The interpreter's low performance is mainly due to the runtime overhead of the bytecode fetch and decode. The PCL data shows the total number of instructions increasing significantly, as one can see in Table **??**. The interpreter also requires extra memory accesses, which tends to increase the percentage of LoadStore instructions and the memory footprint. This may justify the relatively large number of cache misses the interpreter suffers in applications such as Crypt. The hardware seems to be quite good at running the emulation loop: we notice no significant increase in number of stalls compared with the compiled version. The overall stability of the IPC in the emulator is interesting: it ranges from

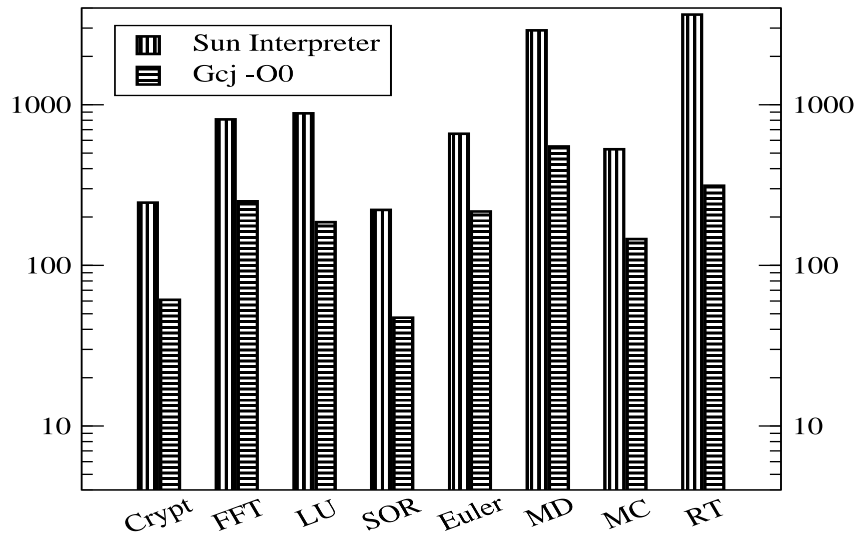

Figure 4: Execution time in seconds.

0.43 to 0.78. On the other hand, the IPC for GCJ varies from 0.29 to 0.91. We believe this is because the emulation process has a strong contribution to the IPC, except in the cases where we need to run external library code.

6.2 Sun JIT Compilation Versus GCJ -O2 Compilation

Next, we compare the performace the Sun JIT and GCJ. We might expect the code generated by static compiler to perform better. In fact, the Sun HotSpot server compiler has better performance than GCJ compiler, for almost every benchmark, as shown in Figure 5.

For the benchmarks FFT, LU, Euler, MolDyn, MC and RT the Sun Server was more efficent than GCJ because of better cache usage, less code generated, less pipeline stalls, and less memory accesses. Only for the benckmarks Crypt and SOR the GCJ compiler obtained better performance. The Sun Client is even more interesting: it tends to perform worse than the Sun Server, as expected, but it still does quite well compared to optimised GCJ.

The impact in time of using a JIT compiler is given by Table 2. The first observation is that compilation time is actually much less than total run-time for these applications, even for GCJ. The second is that GCJ is doing surprisingly well timewise compared to the JIT compilers. The Server compiler is never much faster than GCJ, and in some cases it is much slower. The client compiler is indeed faster than both versions of GCJ, but only up to an order of magnitude. The

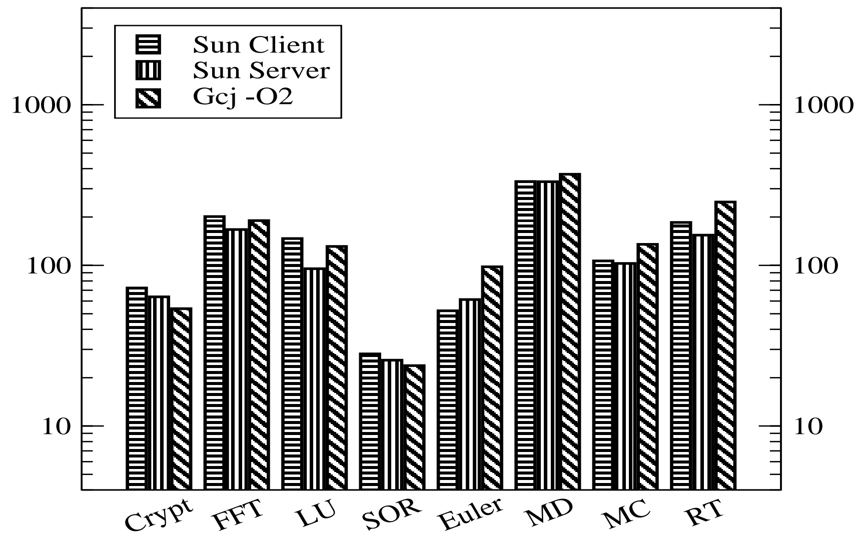

Figure 5: Execution time in seconds.

results suggest that we may need larger applications to extract all advantages of JIT technology.

	SUN		GCJ	
	Client	*Server*	*-O0*	*-O2*
Crypt	0.214	1.296	2.150	3.540
FFT	0.213	1.894	1.640	2.920
LU	0.191	6.247	1.700	3.160
SOR	0.138	1.080	1.530	2.700
Euler	1.627	17.294	3.820	14.390
MolDyn	0.322	18.799	1.750	3.140
MonteCarlo	1.620	33.551	3.710	5.930
RayTracer	0.371	6.257	2.890	4.560

Table 2: Total compilation time in seconds.

A second advantage of JIT is reducing the number of methods compiled, and thus code size, whilst still significantly decreasing the number of bytes executed. Table 3(a) shows that this is indeed the case: the JIT compilers generate much more compact code than GCJ. Table 4 shows the number of methods compiled,

	SUN		GCJ	
	Client	*Server*	*-O0*	*-O2*
Crypt	2167	1328	53810	47354
FFT	2011	2081	52602	47729
LU	1839	1677	54728	47868
SOR	1128	765	47220	43379
Euler	25209	31196	152422	90141
MolDyn	4185	5713	61592	54607
MonteCarlo	13835	8778	124291	116730
RayTracer	3644	1982	76068	71915

(a) Code Size

	Client	Server
Crypt	3.139	18.226
FFT	3.129	3.172
LU	3.123	3.339
SOR	3.119	3.149
Euler	3.328	3.548
MolDyn	3.137	3.170
MonteCarlo	23.527	28.234
RayTracer	3.118	3.163

(b) GC Time

Table 3: Code Size (bytes) and GC Time (seconds).

	Client			Server		
	Bytecodes Executed	*Methods Executed*	*Methods Compiled*	*Bytecodes Executed*	*Methods Executed*	*Methods Compiled*
Crypt	3105533	4317	8	7214242	4317	4
FFT	2815136	4319	14	4557427	4319	24
LU	2454194	4519	12	21050512	4519	16
SOR	1582599	4512	9	6402882	4512	13
Euler	27553472	4609	53	506075419	4614	260
MolDyn	2995555	4876	16	39285436	4876	37
MonteCarlo	5065979	4764	112	41384352	4764	515
RayTracer	2991664	4571	25	27385698	4571	112

Table 4: Sun Bytecodes and Methods.

methods executed and number of bytecode instructions actually executed by the Java compilers. Althougth, the number of methods executed to be the same, it is interesting to notice that the Server compiler tends to execute more bytecodes, which indicates it delays more to optimise a method, but in average ends up compiling more methods. This suggests that the Server compiler generates very compact code. It also suggests that smaller benchmarks may not show the whole story for the Server compiler.

We were intrigued why the Sun compilers did not perform as well in Crypt and SOR. Table 3(b) explains the problem for Crypt: for some reason the server compiler is performing a lot of garbage collection. On the other hand, SOR is

the smallest benchmark, so it is the one where the overheads of JIT are most severe. The tables 5 and 6 show that the Server version performs quite well, as does optimising `GCJ`.

The tables 5 and 6 give some insight into the differences between compilers. Most compilers tend to use the data-structures so the number of cache misses tends to be pretty similar. Sun client and server tend to be particularly close. `GCJ` is often close, but sometimes does significantly worse. The better quality of the Sun server compiler can be seen from the total number of instructions executed. The server compiler does also better than optimised `GCJ` on stalls. This may be from having less instructions, though. Last, the optimised compilers tend to need less Load/Store instructions. Interestingly enough the Sun client compiler is very much better in that respect than `GCJ` without optimisations, and often close to the server version.

		SUN			`GCJ`	
		Interpreter	*Client*	*Server*	*-O0*	*-O2*
Crypt	*Instructions* 10^8	1382.45	620.47	550.34	427.58	360.06
	% LoadStore	99.58	67.86	68.77	86.95	73.77
	Cache Miss 10^8	2.17	0.14	0.14	0.14	0.14
	STALL 10^8	330.93	180.07	180.54	226.43	180.29
	IPC	0.67	1.01	1.02	0.85	0.87
FFT	*Instructions* 10^8	3341.99	501.64	347.70	714.67	334.68
	% LoadStore	100	50.44	43.83	95.46	50.80
	Cache Miss 10^8	69.89	31.88	21.85	28.72	28.38
	STALL 10^8	2535.69	1768.40	1291.12	2093.57	1584.26
	IPC	0.43	0.28	0.24	0.29	0.19
LU	*Instructions* 10^8	3833.98	643.18	227.37	1124.27	482.75
	% LoadStore	100	50.24	36.24	99.60	55.83
	Cache Miss 10^8	20.55	19.06	19.09	18.70	18.76
	STALL 10^8	1102.32	863.68	715.21	1094.51	835.40
	IPC	0.61	0.56	0.29	0.71	0.46
SOR	*Instructions* 10^8	1165.66	164.49	91.87	372.32	127.09
	% LoadStore	100	42.31	43.53	94.85	43.02
	Cache Miss 10^8	21.06	4.08	4.09	4.22	4.16
	STALL 10^8	612.13	136.79	145.24	197.49	135.95
	IPC	0.55	0.65	0.42	0.91	0.61

Table 5: PCL informations.

		SUN			GCJ	
		Interpreter	*Client*	*Server*	*-O0*	*-O2*
Euler	*Instructions* 10^8	3614.42	63.03	247.72	952.15	362.23
	% LoadStore	78.95	63.03	48.68	100	71.15
	Cache Miss 10^8	25.22	8.01	8.01	25.28	19.85
	STALL 10^8	836.32	312.93	307.10	755.99	583.18
	IPC	0.67	0.49	0.54	0.55	0.45
MolDyn	*Instructions* 10^8	14360.11	2022.85	2124.49	3492.74	3037.27
	% LoadStore	79.11	36.78	31.84	72.39	54.53
	Cache Miss 10^8	73.68	52.71	51.90	67.40	67.27
	STALL 10^8	4034.53	1604.17	1663.96	3425.79	3027.90
	IPC	0.62	0.74	0.75	0.60	0.60
Monte Carlo	*Instructions* 10^8	3450.10	692.57	622.29	769.21	644.65
	% LoadStore	76.83	68.43	67.24	75.78	65.84
	Cache Miss 10^8	45.61	2.21	2.13	2.40	5.06
	STALL 10^8	908.53	352.16	337.13	547.07	488.84
	IPC	0.75	0.83	0.79	0.65	0.61
Ray Tracer	*Instructions* 10^8	23769.59	1476.17	1365.61	2390.09	2069.83
	% LoadStore	66.58	58.09	46.88	85.75	74.04
	Cache Miss 10^8	215.56	4.24	3.30	10.99	9.17
	STALL 10^8	3191.44	640.87	693.93	1012.54	608.83
	IPC	0.78	0.95	1.03	0.91	0.98

Table 6: PCL informations.

7 Related Works

Kaffe [Kaffe 2005] is a free software implementation of the Java Virtual Machine. It is designed to be an integral component of an open source or free software Java distribution. Kaffe VM is constantly under development. Unfortunately, Kaffe lacks full compatibility with the current releases of Java, namely, security related features, debugging support, and profiling support.

The Kaffe JIT compiler compiles a method at a time as it sees them. The first time a method is called in the bytecode, the method is translated into native code and cached, also, the dispatch table is updated. Next time, the program will jump directly to the previously translated native code. The compiler does not perform any global optimizations, all optimizations are local per basic block. This project intends to allow software reuse when porting compiler code to a new architecture, which results in more rapid and cost-effective code development.

The IBM compiler [Suganuma et al 2000] implements adaptive optimization. In a fashion similar to the HotSpot compiler, it first runs a program using an

interpreter, detecting the critical *hot spots* in the program as it runs. It further monitors program *hot spots* continuosly as the program runs so that the system can adapt its performance to changes in the program behavior.

Experimental results [Suganuma et al 2000] have been shown that the optimizations used by this compiler are very effective for several types of programs. Overall, the IBM JIT compiler combined with IBM's enhanced JVM is widely regarded as one of the top performing Java execution environments.

The Jikes Research Virtual Machine (RVM) compiler [Jikes 2005, Alpern et al 2000], also developed by an IBM team, translate Java bytecodes to machine code at runtime. For this propose, the runtime system can use one of three diferent, but compatible, compilers, namely, the baseline compiler, the optimizing compiler, and the quick compiler.

The baseline compiler provides a transparently correct compiler. This compiler is then used as a reference in the development of the RVM. However, it does not generate high-performance target code.

The optimizing compiler applies traditional static compiler optimizations to obtain high-quality machine code, as well as a number of new optimizations that are specific to the dynamic Java context. The cost of running the optimizing compiler is too high for it to be profitably employed on methods that are only infrequently executed. The optimizing compiler intended to ensure that Java bytecodes are compiled efficiently. Its goal is to generate the best possible code for the selected methods.

The quick compiler compiles each method as it executes for the first time. It balances compile-time and run-time costs by applying a few highly effective optimizations. This compiler tries to limit compile time by an overall approach of minimal transformation, efficient data structures, and few passes over the source code.

The RVM has not yet implemented a comprehensive strategy to select best compiler for each method. Switching from the quick to the optimizing compiler will be done based on runtime profiling.

8 Conclusions

We evaluated three different Java implementation technologies on a set of well-known benchmarks. Our main conclusion is that JIT technology performs very well: it introduces very significant improvements over emulation, and it performs very well compared to a traditional compiler. JIT technology preserves the main advantages of bytecode, such as portability and compactness. And compiling from bytecode does not seem to introduce significant overheads. JIT technology thus seems to be doing well at combining the advantages of both worlds.

We were somewhat disappointed by the results obtained by `GCJ`. Compilation time is not the problem: `GCJ` does close to Sun's server compiler, although much

worse than the client compiler. Unfortunately, GCJ seems to generate worse quality code than the Sun Server compiler, and often than the Sun-Client compiler. Namely, for some applications GCJ has a much worse miss-rate, indicating issues with variable allocation.

In all cases compilation time was a negligible fraction of total running time. This might suggest skipping the Sun Client compiler altogether, at least for these applications. On the other hand, the Sun Client compiler seems to be better at compiling less methods with close to the same performance. This suggests that an interesting research step would be to study the performance of the Server compiler under the same JIT activation parameters: we would expect a significant saving in code size with low costs in running time, which might interesting for embedded applications. It would also be interesting to compare the clients with other state of the art compilers, such as the IBM compilers [Suganuma et al 2000], and on a wider range of applications and hardware.

References

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

[Arnold et al. 2000] Arnold, Ken, Gosling, James, Holmes, David: "The Java Programming Language"; Addison Wesley, California (2000).

[Berrendorf et al 2003] Berrendorf, Rudolf, Ziegler, Heinz, Mohr, Bernd: "Performance Counter Library"; (2005), http://www.fz-juelich.de/zam/PCL/.

[Bodik et al 1997] Bodik, Rastislav, Gupta, Rajiv, Soffa, Lou Mary: "Interprocedural conditional branch elimination"; Proc. of the ACM SIGPLAN 1997 conference on Programming language design and implementation; (1997), 146-158.

[Briggs et al 1997] Briggs, Preston, Cooper, Keith D., Simpson, L. Taylor: "Value Numbering"; Software emdash Practice and Experience 27, 6(1997), 701-724.

[Detlefs et al 1999] Detlefs, David, Agesen Ole: "Inlining of Virtual Methods", Proc. of the 13th European Conference on Object-Oriented Programming, (Jun 1999).

[Gulwani et al 2004] Gulvani, Sumit, Necula, George C.: "Global value numbering using random interpretation"; Proc. of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, (2004), 342-352.

[Fink at al 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), 421-252.

[GCC 2005] GNU Team: "GNU Compiler Collection"; (2005), http://gcc.gnu.org.

[GCJ 2005] GNU Team: "GNU Compiler for Java"; (2005), http://gcc.gnu.org/java.

[GIMPLE 2005] GNU Team: "GIMPLE"; (2005), http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.

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

[Huang et al 1999] Huang, J. C., Leng T.: "Generalized Loop-Unrolling: a Method for Program Speedu-Up"; Proc. of the IEEE Workshop on Application-Specific Software Engineering and Technology, (Mar 1999), 244-248.

[Jikes 2005] IBM Team: "Jikes Research Virtual Machine"; (2005), http://www-124.ibm.com/developerworks/oss/jikesrvm/.

[JGF 2005] Java Grande Forum: “Java Grande Forum Benchmarck”;(2005), http://www.epcc.ed.ac.uk/javagrande/.

[Kaffe 2005] Kaffe Team: ”Kaffe Virtual Machine”; (2005), http://www.kaffe.org/.

[Krall 1998] Krall, Andreas: “Efficient JavaVM Just-in-Time Compilation”; International Conference on Parallel Architectures and Compilation Techniques, Paris, France (1998), 205-212.

[Kawahito et al 2000] Kawahito, Hideaki Komatsu, Nakatani, Toshio: ”Effective Null Pointer Check Elimination Utilizing Hardware Trap”; ACM SIGARCH Computer Architecture New 28, 5(2000), 139-149.

[Lindhom et al 1999] Lindholm, Tim, Yellin, Frank: “The Java Virtual Machine Specification Second Editon”, Addison Wesley, California, USA (1999).

[Paleczny et al 2001] Paleczny, M., Vich, C., Click C.: ”The Java HotSpot Server Compiler”; Proc. of the Java Virtual Machine Research and Technology Symposium, (Apr 2001), 1-12.

[Pelegri-Llopart et al 1988] Pelegri-Llopart, E., Graham, S. L.: ”Optimal Code Generation for expression Trees: An Application BURS Theory”; Proc. of the Conference on Principles of Programming Languages, Sao Francisco, USA, (Jun 1988), 294-308.

[Plezbert et al. 1997] Plezbert, Michael P., Cytron, Ron K.: “Does Just-In-Time = Better Late Than Never?”; Proc. of the Symposium on Principles of Programming Language, Paris, France (Jan 1997), 120-131.

[Romer et al 1996] Romer, Theodore H., Lee, Dennis, Voelker, Geffrey M., Wolman, Alec, Wong, Wayne A., Baer, Jean-Loup, Bershad, Brian N., Levy, Henry M.: “The structure and performance of interpreters”; Proc. 7th International Conference on Architectural Support for Programming Languages and Operating System, ACM, New York USA (1996), 150-159.

[Simpson 1994] Simpson, T.: ”Global Value Numbering”, Rice University, (1994), urlftp://cs.rice.edu/public/preston/optimizer/gval.ps.

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

[Sun 2003] Sun MicroSystems: “The Java HotSpot Virtual Machine”. Technical Report, Sun Developer Network Community, (2003).

[Vernners 1999] Venners, Bill: ”Inside the Java 2 Virtual Machine”; Mc Graw Hill, New York, USA (1999).

[Wedign et al 1984] Wedign, Robert G., Rose, Marc A.: ”The Reduction of Branch Instruction Execution Overhead using Structured Control Flow”; Proc. of the 11th annual international symposium on Computer architecture, (1984), 119-125.

[White Paper 2004] White Paper, ”Inlining Virtual Methods White Paper”, (2005), http://whitepapers.zdnet.co.uk/0,39025942,60015778p,00.htm.