LaTTe: A Java VM Just-In-Time Compiler with Fast and Efficient Register Allocation[*]

Peter Molnár

January 7, 2007

Abstract

Higher Java performance for network computing on desktop machines can be achieved by a Just-in-Time compiler that translates the stack-based Java byte code into machine code. This document presents the Just-in-Time compiler approach taken by the LaTTE virtual machine. Special focus is taken on the fast and efficient register mapping and allocation algorithm implemented in LaTTe.

1 Introduction

A Java *just-in-time* compiler is a component of a Java virtual machine that in contrast to static, off-line translation [2] translates byte code to native code immediately prior to its execution. Just-in-time compilation is more suitable for Java due to its dynamic loading nature.

The main challenge of Java just-in-time compilation is the generation of efficient code by allocating stack slots and local variables to registers efficiently. As the compilation time becomes part of the running time of the code, the register allocation requires a fast algorithm. Conventional algorithms based on graph-coloring are rather unsuitable for a JIT compiler.

This work presents LaTTe [1], a just-in-time compiler for the SPARC platform. LaTTe translates Java byte code into intermediate code using symbolic registers. A fast and efficient algorithm then allocates real registers to the symbolic registers, coalescing unnecessary copy operations representing pushes and pops to and from the operand stack.

2 Java virtual machine and SPARC

The Java virtual machine is a typed stack architecture [2]. A thread of execution has a Java stack associated where a new activation record is pushed each time a method is invoked and popped upon method return. The activation record includes *local variables* and an *operand stack* used to perform computations.

The calling convention for Java method calls looks like follows: the arguments are pushed on the callers operand stack, including the `this` pointer as first argument. They are popped by the JVM

[*]This document is intended to be a summary of [1]. The ideas and presented algorithms are the work of the authors of [1].

and set up as first local variables of the called method. When the method returns, the return value is transfered from the top of the callee's operand stack to the top of the caller's operand stack.

The SPARC is a RISC architecture featuring a register-based instruction set [3]. A function uses a register window consisting of 24 registers partitioned into 8 in-registers (`%i0 - %i7`), 8 local registers (`%l0 - %l7`) and 8 out-registers (`%o0 - %o7`). When a function is called, the register window is rotated, so that the caller's out registers correspond to the callee's in registers. The callee saves the return value to in-register 0, which corresponds to out-register 0 from the view of the caller.

3 Code generation

In LaTTe, native code is generated on a per-method basis. The translation process is started when a Java method is called for the first time and it consists of five passes. First control join points and subroutines are identified in the code. Next a control flow graph (CFG) is computed containing instructions with symbolic registers. In the following optional stage optimisations are performed. In the fourth stage a fast register allocation algorithm is performed and finally native code is emitted. The rest of this section focuses on passes two and four.

3.1 Translation of byte code to pseudo code

The SPARC instruction generated by a specific byte code instruction is determined only by the opcode. Instead of registers as operands the instruction uses symbolic registers. These symbolic registers are composed of three parts:

- type: `a` = address, `i` = integer, `f` = float, `l` = long, `d` = double
- location: `s` = operand stack, `l` = local variable, `t` = generated temporary
- number further distinguishing the symbolic register

An additional `TOP` compile-time variable is used to address the topmost item on the stack. For example if the stack size is 4 then `is{TOP - 1}` is equivalent to `is3`. The types of the variables in the stack slots are stored in a compile-time array `type[1 .. TOP]`. LaTTe traverses the code in depth-first order. If a byte code instruction that pushes elements on the stack is encountered, `TOP` is incremented by the number of elements pushed. Byte codes instructions popping stack elements decrement `TOP`.

The stack computational model of the JVM leads to the consequence that byte code instructions transferring values between local variables and the stack occur frequently. These `load` and `store` instructions are translated to `mov` instructions with pseudo registers representing stack slots and local variables as operands. An example is shown in table 1.

Bytecode	Pseudo SPARC Code	Compile time actions	Stack
`iload n`	`mov il{n}, is{TOP+1}`	`TOP += 1, type[TOP] = i`	`...  => ..., local variable n`
`astore n`	`mov as{TOP}, al{n}`	`TOP -= 1`	`..., value => ...`

Table 1: Translation of load and store instructions

Arithmetic byte code instructions operate of the top items of the operand stack and are directly mapped to pseudo instructions. An example is shown in table 2.

Bytecode	Pseudo SPARC Code	Compile time actions	Stack
`iadd`	`add is{TOP-1}, is{TOP}, is{TOP-1}`	`TOP -= 1`	`..., x, y => ..., (x+y)`

Table 2: Translation of arithmetic instructions

A memory representation of an object in the LaTTe JVM contains a pointer to the virtual/interface table, a 32 bit lock followed by the instance data. The object fields are located at a specified offset from the start of the object. Field access is mapped to load and store pseudo instructions. An example is shown in table 3.

Bytecode	Pseudo SPARC Code	Compile time actions	Stack
`getfield x.foo`	`ld as{TOP} + offset, is{TOP}`	`type[TOP] = i`	`..., object => ..., integer`
`putfield x.foo`	`st is{TOP}, as{TOP-1} + offset`	`TOP -= 2`	`..., object, integer => ...`

Table 3: Translation of filed access instructions

For each loaded class a *virtual function table* is maintained. The table contains the start address of each method defined in the class or inherited from the superclass. Because of single inheritance in Java, if a method of a class is placed at offset `n` in the virtual function table, it can be placed at offset `n` in the virtual function tables of all subclasses. The symbolic registers corresponding to the arguments and return values are allocated in the register allocation stage to registers according to SPARC calling conventions. Consult table 4 for an example.

Byte code	`invokevirtual x.func // (int, int) -> int`
Stack	`..., object, argument, argument => ..., result`

Pseudo code	`ld as{TOP-2}, at0`
	`// pointer to is located at the beginning of the object`
	`ld at0 + function offset, at1`
	`// address of method is located at a specified offset in the table`
	`call at1`

Preferred register allocation	`as{TOP-2}:%o0, is{TOP-1}:%o1, is{TOP}:%o2`
	`is{TOP-2}:%o0`

Compile time	`TOP -= 2, type[TOP] = i`

Table 4: Translation of method invocation

3.2 Fast register allocation

The CFG of pseudo SPARC instructions is partitioned into tree regions which are single-entry, multiple-exit subgraphs shaped like trees (same as *extended basic blocks* [4]). Tree regions start at the beginning of the program or at control join points and end at the of the program or at other join points. Next, last uses of symbolic registers are calculated. Stack and temporary registers are supposed to be dead after they are used in an instruction, so determining their life range is rather easy. For local symbolic registers an upper bound of their liveness is computed. The regions are then register-allocated one by one in *reverse post-order* traversal of the CFG of regions.

Each region is traversed twice. Once in a *post-order* traversal called the *backward-sweep* and once in a *depth-first traversal* called the *forward sweep.* The backward sweep determines the preferred destination registers for instructions. The forward sweep uses this information to perform the actual register allocation. During each traversal, a map which is a set of (symbolic, real) register pairs is constructed. For the backward sweep it is called *p_map* while for the forward sweep is called *h_map* .

The purpose of the backward sweep algorithm is to determine the *p_map* for destinations of instructions based on the required register assignment at the end of the region, or at method call/returns according to the conventions. For example if at the end of a region a symbolic register `il2` is allocated to register `r` and there are the instructions `add is1, is2, is1; mov is1, il2`, then the destination `is1` of the `add` is preferred to be allocated to `r` to avoid a copy. If `x` is preferred to be allocated to `r` under a `mov y, x`, then `y` is preferred to be allocated to `r` above the instruction.

The forward sweep algorithm computes the *h_map* starting of an initial *h_map* . It maintains `refcount`, a map showing the count of symbolic registers mapped to a real register and `freereg`, the set of real registers not yet mapped to real registers. In a depth-first traversal of the region, if an instruction `z = x + y` is encountered, first the right hand side is generated as `h[x] + h[y]`. If `x` is the last use, the `refcount` of `h[x]` is decremented. If it even gets zero `h[x]` is added to `freereg`. The target register `z` in case of a copy instruction `z = x` is allocated to the same register as `x`. Otherwise if there is a preferred register for `z` and the register is available in `freereg`, it is chosen. In case it is not available in `freereg` and arbitrary free register is chosen. The pair (z, chosen register) is then added to `h`.

If, in the traversal, we encounter the root of the next region, we save the current *h_map* at that root, so that it can be used as the initial *h_map* of that forward sweep. As the root is a join point, it can be reached by more than one forward sweeps. If it was already reached and therefore has already an initial *h_map* , we need to reconcile the old initial *h_map* with the new initial *h_map* .

Let's call *h_old* the initial *h_map* at a region and *h_new* the new initial *h_map* to be set in the forward sweep. If `h_old[x]=h_old[y]=r` and `h_new[x]=h_new[y]=r'` then a copy `r=r'` has to generated on the new incoming edge. This will preserve the old mapping, `h[x]=h[y]=r`. However, if `h_new[x]=r'` and `h_new[y]=r''` there is a problem. Although the mappings can be reconciled inserting copies also in the old incoming edge, the old mapping `h=h_old` on the regions root can not be preserved any more. If the region has not yet been register allocated, this is no problem. But if the region has already been allocated using the *h_old* mapping, it has to be reallocated.

If during forward sweep no registers are available on an instruction I, a real register for spilling is chosen based on a heuristic. Assuming that `h[x] = h[y] = r` and `r` is chosen for spilling, we generate a store instruction before I `st x, SPILL0`. Than we mark `x` and `y` last uses. By register allocating the generated store, we allocate `st r, SPILL0`. The symbolic registers `x` and `y` are than mapped to `SPILL0` (`h[x] = h[y] = SPILL0`). If a spilled register is used later, a load is inserted before.

4 Comparison with previous JIT compilation techniques

In this section the LaTTe JIT compiler is compared with JVMs whose compilation techniques were published including Kaffe [5], VTune [6] and CACAO [7].

Kaffe features a relatively simple single-pass JIT. In Kaffe, local variables and stack slots are mapped to the C stack of the translated method. If a variable or stack slot is used in a basic block, it is loaded to a register and spilled back at the end of the basic block. The consequence are many loads and stores in the generated code. Kaffe generates native code in its single pass, without generating any intermediate code.

Intel's VTune includes a JIT compiler for the x86 platform. Here, local variables are globally preallocated before the translation starts. Then code is generated in a single pass with local register allocation for stack slots and temporaries. To avoid copy operations, a compile-time *mimic stack* is computed. If at basic block boundaries the mimic stack is found to be not empty, all stack slots are spilled to the C stack.

In CACAO, local variables are preallocated like in VTune. Stack slots that are live beyond a basic block are allocated to *interface pseudo registers*. CACAO first converts the byte code to an intermediate representation creating a compile-time *static stack* which contains information similar to symbolic registers in LaTTe. Than an elaborate stack analysis is used to eliminate copy instructions.

The preallocated local variables in CACAO and VTune lead to inefficiency problems compared with LaTTe. First, if a local variable is copied to another, in LaTTe they can both be allocated to the same register. Second, if local variables have different life ranges, then in LaTTe they can be allocated to the same register.

The fact that CACAO an VTune give up coalescing at basic block boundaries if the compile time stack is not empty represents another inefficiency. LaTTe in contrast resolves conflicts at basic block boundaries and generates less instructions.

5 Summary

In this document LaTTe, a Java JIT compiler featuring fast and efficient register allocation has been introduced. The algorithm used represents a solution that trades off quality and speed of register allocation such that it's suitable for JIT compilation. This is confirmed by experimental results in [1].

References

[1] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, Seungil Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, Erik R. Altman, "LaTTe: A Java VM Just-In-Time Compiler with Fast and Efficient Register Allocation," in International Conference on Parallel Architectures and Compilation Techniques, October 1999

[2] F. Yellin and T. Lindholm, *The Java Virtual Machine Specification*, Addision-Wesley, 1996.

[3] D. L. Weaver and T. Germond, *The SPARC Architecture Manual Version 9*, 1994

[4] R. Morgan, *Building an optimising compiler*, Digital Press, 1998

[5] T. Wilkinson, "Kaffe: A JIT and Interpreting Virtual Machine to Run Java Code, " http://www.transvirtual.com/, 1998

[6] A.-R. Adl-Tabatabai et. al. "Fast, Effective Code Generation in a Just-In-Time Java Compiler," in *Proceedings of ACM PLDI '98*, Jun 1998

[7] A. Krall, "Efficient JavaVM Just-in-Time Compilation," in *Proceedings of PACT'98*, 1998