Reducing Stack Usage in Java Bytecode Execution

Wes Munsil and Chia-Jiu Wang

Department of Electrical and Computer Engineering
University of Colorado at Colorado Springs
P. O. Box 7150
Colorado Springs, CO 80933 - 7150

Abstract

For many years, the Tomasulo method of dynamically scheduling instructions for execution in a load/store processor has been known and used. This paper presents an adaptation of the Tomasulo method to a stack-based processor architecture, and illustrates its use in a software simulator of a subset of the Java Virtual Machine. Experimental results show that the adapted Tomasulo method reduces stack usage, in some cases eliminating it altogether. This method should be of interest to computer architects and those involved in the implementation and use of the Java programming language.

1. Introduction

In the 1960s, R. M. Tomasulo and others developed and published a method to achieve concurrent execution of floating point instructions in the IBM System/360 Model 91 computer. This method provided automatic, efficient exploitation of multiple execution units by means of buffers called reservation stations and a common data bus (CDB). The Tomasulo method has been described at length in the literature ([1], [2]). This paper describes an adaptation of the Tomasulo method from its original domain of register-based architectures to stack-based architectures, and shows an implementation of the adapted method in a software simulator of a subset of the Java Virtual Machine ([3], [4], [5], [6]). The adapted method reduces stack usage, and in some cases effectively eliminates usage of the stack altogether. In this respect, it is similar to the instruction folding technique of Java bytecode execution described elsewhere ([7]).

The programming language Java has recently generated much interest in the technical community, primarily because of its association with Internet ([8]). Proponents tout a number of virtues of Java that enhance its suitability for internetworking environment: object orientation, multithreading support, socket support, and so on. But the single most important attribute of Java from this perspective is an implementation detail that makes compiled Java classes portable across multiple platforms: namely, that Java is typically compiled, not into native machine code, but into Java bytecodes, the machine code for an abstraction called the Java Virtual Machine. This Virtual Machine is simulated on many platforms: thus, compiled Java programs can be downloaded to any client node and executed (via simulation) there. For security reasons, downloaded bytecodes are subject to an extensive series of consistency checks by a software component called the Java Bytecode Verifier prior to their simulated execution; this guards against unauthorized type conversions, undisciplined pointer usage, and a host of other questionable or insecure practices.

But Sun Microsystems, Java's proud parent, has even more ambitious plans for a series of Java processors. In its picoJava, microJava, and ultraJava lines, Sun and its licenses LG Semicon, Mitsubishi, and Samsung Electronics have committed to implement the Java Virtual Machine in silicon, and make these JavaChips significant players in the embedded microprocessor and microcontroller market.

The Java Virtual Machine is stack-based. It uses a stack for expression evaluation and argument passing. Now reduction of stack usage is an important goal in stack-based architectures, because the stack will consume ample silicon areas. So application of the adapted Tomasulo method to reduce stack usage of the simulated execution of Java bytecodes is appealing.

We begin by discussing related work, namely, the Tomasulo method itself, and the instruction folding technique used in JavaChips. We then describe our overall technical approach and our design rationale of the adapted Tomasulo method. Finally, we present and discuss our experimental results, and point toward possible future work in this area.

2. Related Work

We describe two related areas of work that served as inspiration for this paper: the Tomasulo method [1] [2] and certain stack optimizations in Sun's picoJava I processor architecture [7].

Hennessy and Patterson [2] present a detailed exposition of the Tomasulo method based on Tomasulo's original paper [1]. The idea is to allow instructions to execute out of order by attacking the "register bottleneck". Operands need not come from registers; they can come directly from the functional units at which they are produced, by means of buffers called reservation stations associated with the functional units. (Load and store operations are performed by load and store functional units respectively, just as other operations are performed by traditional functional units.) Instructions are issued to reservation stations and await their operands there. Reservation stations thus function as a set of virtual registers, and instruction executions that would otherwise have to stall due to various data hazards can proceed. This use of reservation stations instead of registers as operand sources is commonly called register renaming.

The crux of the Tomasulo method is the representation of operands within a reservation station. Each operand is represented by an ordered pair <Q, V>, where Q and V are defined as follows:
- if the operand has already arrived at the reservation station, then Q is blank and V is the value of the operand; otherwise,
- Q identifies the reservation station that is to provide the operand.

An instruction can be issued to a reservation station as soon as one (that is, one reservation station) is available, but it can not begin execution until all its operands are available. When a functional unit completes execution of an instruction, it broadcasts the result over the Common Data Bus (CDB); any functional units awaiting an operand from that functional unit clear their Q field and read V off the CDB. Our work adapts the Tomasulo method to stack-based architectures, in which we can capitalize on known regularities in operand usage to attain, not only more parallelism, but also less stack usage.

Sun [7] has described a variety of implementation techniques in their picoJava I core processor architecture that are geared toward removing some of the inefficiencies in a stack-based architecture. One of these, instruction folding, is based on the realization that, "frequently, an instruction that copies data from a local variable to the top of the stack is followed immediately by an instruction that consumes the data." When this situation arises, the picoJava instruction decoder effectively folds these two instructions together. This compound instruction performs the operation as if the local variable were already located at the top of the stack. Sun reports that the folding operation eliminates up to 60% of the inherent inefficiency in the stack architecture. Our work attempts to regularize, generalize, and extend this folding optimization, by casting it in the framework of the Tomasulo method.

3. The Adapted Tomasulo Method

We now describe the manner in which the Tomasulo method can be adapted to a stack-based architecture. Consider first a simple example. Suppose we have a stack-based machine S, with traditional instructions push, pop, add, mult, and so on. The following S instructions perform the assignment a = b + c. The stack is initially empty and the stack height prior to the execution of each instruction is represented as t.

t	S instruction
0	push b
1	push c

```
2       add
1       pop     a
```

As these instructions are executed, the stack behaves as follows as shown in Figure 1.

Figure 1. stack contents after the execution of each instruction

Now we consider the stack to be sequentially named registers as illustrated in Figure 2.

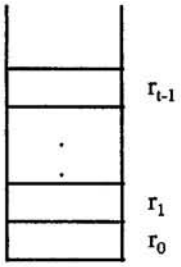

Figure 2 . Stack as sequentially named registers

The stack behavior is shown in Figure 3 where stack is viewed as a set of named registers

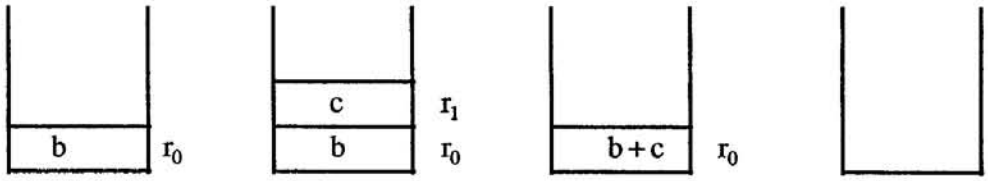

Figure 3. Stack behavior when viewed as named registers

Suppose we have a load/store machine LS whose instruction set uses these registers as operands. The above sequence of S instructions is then equivalent to the following sequence of LS instructions.

```
t       LS instruction

0       load r0, b
1       load r1, c
2       add r0, r1, r0
1       store r0, a
```

We can make several observations about the LS instructions in the code sequence:
1. Load instructions always load r_t
2. Store instructions always store r_{t-1}. Furthermore, the value in r_{t-1} is unneeded after the store.
3. Binary operations always combine r_{t-1} and r_{t-2} into r_{t-2}. As with stores, the value in r_{t-1} is unneeded after the operation.

These observations about this simple example may be shown to be true in general. We do not prove this, but we do provide further evidence in the form of a slightly more complicated example, this time for the assignment a = (b+c) * (d+e)

t	S instruction		LS instruction	
0	push	b	load	r_0, b
1	push	c	load	r_1, c
2	add		add	r_0, r_1, r_0
1	push	d	load	r_1, d
2	push	e	load	r_2, e
3	add		add	r_1, r_2, r_1
2	mult		mult	r_0, r_1, r_0
1	pop	a	store	r_0, a

Now we have the key insights needed in the adaptation of the Tomasulo method. The fact that the value in r_{t-1} is unneeded after stores and binary operations allows the adapted Tomasulo method to clear r_{t-1}'s Q field. This allows the processor to suppress register usage (i.e., stack usage) in some cases, and occasionally to eliminate it altogether. We have, in essence, replaced elements of the stack by the virtual registers that are the reservation stations.

To adapt the Tomasulo method to a stack-based architecture, we first map the elements of the stack in the natural way onto sequentially named registers in an equivalent load/store architecture, and then apply the Tomasulo method, keeping track of the stack height t, with the additional actions of clearing r_{t-1}'s Q field after stores and binary operations.

An objection to the foregoing will by now have occurred to the astute reader. We have tacitly assumed that the value of t is well-defined: in other words, that the stack height at the beginning of execution of any instruction is the same regardless of the path by which control arrived at that instruction. This is indeed a limitation of this technique, for it is certainly possible in general to construct sequences of stack-based instructions for which this assumption is violated. There are two responses to this objection, one weak and the other strong. The weak response is that such sequences would violate our intuitive notion of the "well-formedness" of stack-based programs. The strong response is that, for our application area of Java Virtual Machine simulation, the assumption is in fact a requirement: the Java Bytecode Verifier enforces this requirement, among others.

4. Simulation Results

We tested six Java programs. Program 1 does a simple x = a + b. Program 2 does a simple x = a * b. Program 3 does x = (a + b)*(c + d). Program 4 does x = a + b - (c + d -(e + f - (g + h - j))). Program 5 does x = a + b * (c + d * (e + f * (g + h * j))). Program 6 does x = (a + b) * (c + d); y = (a - b) / (c * d) * (e + f). All numbers are floating point numbers in all six programs. A C++ simulator is written and can be parameterized with certain aspects of the simulation architecture, namely, the number of various kinds of floating point functional units and the latencies of various floating point operations. For results in this paper, we used the following values:

Size of the floating point unit instruction queue:	6 instructions
Number of floating point loaders:	6
Floating point load latency:	4 cycles
Number of floating point stores:	3
Floating point store latency:	2 cycles
Number of floating point adders:	3
Floating point add latency:	2 cycles
Floating point subtract latency:	2 cycles
Number of floating point multipliers:	2
Floating point multiply latency:	10 cycles
Floating point divide latency:	40 cycles

Table 1. Simulation results for all six programs

Program	Stack height	Stack height with adapted Tomasulo	Savings (%)
program 1	2	0	100
program 2	2	0	100
program 3	3	0	100
program 4	5	3	40
program 5	9	9	0
program 6	3	1	67

From simulation results, we observe that for simple cases, such as the first three programs, stack usage is eliminated altogether. When stack usage is high, the saving is respectable for low-latency operations (the fourth program), but nonexistent for high-latency operations (the fifth program). The saving is good for typical instruction mixes (the sixth program).

5. Conclusions and Future Work

We have presented the adapted Tomasulo method for stack-based architectures. We add one more step in the Tomasulo method which is to clear r_{t-1}'s Q field after stores and binary operations. Experimental results show that the saving is good for programs with typical instruction mixes. This saving is important when committing stack-based processor such as Java, into silicon.

The work described in this paper can be continued in several directions: (1) Support branch prediction and speculation and simulate a reorder buffer; (2) Deal with other Java Virtual Machine bytecodes; (3) Investigate the dependence of the results on the numbers of functional units and latencies of operations; (4) Investigate the dependence of the results on the instruction mix.

References:

[1] Tomasulo, R. M. "An Efficient Algorithm for Exploiting Multiple Arithmetic units", IBM Journal of Research and Development, Vol II, pp. 25 - 33, 1967.

[2] Hennessy, J. L. and D. A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, Inc.; 1996.

[3] Sun Microsystems, Inc., The Java Virtual Machine Specification, Release 1.0 Beta, Draft; http://www.javasoft.com/doc/language_vm_specification.html; August 21, 1995.

[4] Lindholm, T. and F. Yellin; The Java Virtual Machine Specification; Addison-Wesley Publishing Company; 1996.

[5] Venners, B.; Under the Hood: The Lean, Mean, Virtual machine; http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html; June, 1996.

[6] Venners, B.; Under the Hood: The Java Class File Lifestyle; http://www.javaworld.com/javaworld/jw-07-1996/jw-07-classfile.html; July, 1996.

[7] Sun Microsystems, Inc.; picoJava I Microprocessor Core Architecture; http://www.sun.com/sparc/whitepapers/wpr-0014-01.

[8] Kramer, D.; The Java Platform, A White Paper; http://www.javasoft.com/doc/whitePaper.Platform/CreditsPage.doc.html; May, 1996.