

Reducing loads and stores in stack architectures

Thomas VanDrunen Antony L. Hosking Jens Palsberg
Purdue University, Dept. of Computer Science
West Lafayette, IN 47907-1398, USA
{vandrutj,hosking,palsberg}@cs.purdue.edu

September 30, 2001

Abstract

The stack model of execution uses a stack to hold temporary results during evaluation of a program. Systems such as Java virtual machines that use this model can be implemented so that they have more efficient access to the stack than to local variables. Thus, converting local variable accesses into stack accesses can improve the performance of stack-based programs, as suggested by the experiments of Koopman [1994], Maierhofer and Ertl [1998], Vallée-Rai et al. [2000], and Shpeisman and Tikir [1999]. In this paper we provide a foundation for these experiments by formalizing and generalizing various known transformations for reducing the number of loads and stores, and by proving their correctness.

1 Introduction

Stack architectures were first formulated in the 1960s as a response to a belief that it is hard for compilers to utilize registers effectively. Instruction sets were based on a *stack model* of execution, in which operands are pushed on a stack from memory or popped off the stack into memory. Operations take their operands from the stack and then place the result back onto the stack. In addition to simplifying compilers by eliminating register allocation, stack machines permit a compact instruction encoding, since an instruction needs only to encode the operation to be performed, while its operands and result are implicitly consumed from and produced to the stack, respectively. Meanwhile, advances in register allocation have seen modern hardware revert to general-purpose register architectures. This has been taken to the extreme with current load-store RISC architectures, in which operations consume/produce operands solely from/to registers, while load and store instructions move values between memory and registers.

The advent of network computing has prompted renewed interest in stack architectures as a compact encoding for programs that must be transmitted across the Internet, since smaller programs take less time to transmit. For precisely this reason, the Java Virtual Machine (JVM) specification [Lindholm and Yellin 1999] defines an abstract execution platform to which Java programs are compiled, using a stack-based instruction set. These are commonly referred to as *bytecode* instructions since they are encoded as a byte stream with each instruction represented as a variable length sequence of bytes. A *Java virtual machine* is a software program that executes compiled Java bytecode programs. It can either decode the bytecodes and interpret their effects, or it can dynamically translate them to the native instruction set of the host machine for direct execution. Such dynamic translation of Java bytecodes to native host instructions is commonly referred to as *just-in-time* (JIT) compilation, since the bytecode programs are translated as necessary on-the-fly, as they are executed. Some virtual machine implementations even mix these execution strategies, interpreting bytecode that is infrequently executed and for which the translation overhead is needlessly expensive, while compiling frequently executed bytecode for which translation up front pays off because the compiled native code can be cached for subsequent re-execution.

In any case, whether bytecode is interpreted or compiled, efficient execution requires mapping virtual machine resources to those of the underlying hardware. In particular, both source-level local variables and intermediate values held on the JVM stack are candidates for allocation to hardware registers. When bytecode is interpreted, a common technique is to manage an explicit evaluation stack in memory, and to cache values from the top of the stack in hardware registers [Ertl 1995], while locals are usually stored in memory. As such, any reduction in the use of locals in favor of stack slots can reduce memory traffic and increase performance: instead of loading a variable from memory onto the stack each time it is used, its value can be cached in the stack and reused in subsequent operations.

When bytecode is translated to native code by a JIT compiler, there are many register allocation techniques that can be applied to allocating both locals and stack slots to hardware registers. A key issue in dynamic translation is keeping compile times to a minimum (including the time for register allocation) if “just-in-time” is not to become “just-too-late”. Again, several allocation techniques are possible. A simple approach is to assign a fixed register to each stack slot (handling overflow where necessary), and to use register renaming to simulate stack manipulation operations such as those supported by Java in the form of the `swap` and `dup` bytecodes. More advanced bytecode compilers will pool stack slots and local variables and use traditional register allocation techniques based on polynomial-time approximations to the graph coloring problem [Chaitin 1982; Chow and Hennessy 1990; Briggs et al. 1994; George and Appel 1996], or more recent approaches that offer a near-linear approximation [Poletto and Sarkar 1999] in an attempt to be more competitive in performance-sensitive settings such as JIT compilation. Again, eliminating locals in favor of stack slots may improve performance by reducing register pressure and easing the job of the allocator.

The task of generating bytecode that effectively uses the evaluation stack to both minimize code size and improve performance by avoiding local variables has been referred to as *stack allocation* [Maierhofer and Ertl 1998]. It is made even more difficult in the face of common techniques for scalar optimization [Muchnick 1997]. These often rely on code transformations that effectively assume an underlying register machine, such that temporary values can usually be held in registers. For example, common subexpression elimination (CSE) relies on caching the result of the first evaluation of a redundant expression in a temporary, which can then be reused at subsequent occurrences of the redundant expression instead of recomputing it. A similar situation holds for variations on this theme, such as partial redundancy elimination (PRE) [Morel and Renvoise 1979; Chow et al. 1997]. Moreover, commonly-used intermediate representations for optimizing compilers, such as static single-assignment form (SSA) [Cytron et al. 1991; Stolz et al. 1994; Gerlek et al. 1995; Wolfe 1996; Chow et al. 1997; Briggs et al. 1998; Hasti and Horwitz 1998; Sastry and Ju 1998] also assume that temporaries can be freely created, with lifetimes that do not conform to a last-defined-first-used stack protocol. Generating stack-based code from such intermediate representations requires effort to avoid allocating temporaries as local variables, instead allocating them on the stack wherever possible.

Stack allocation implies that operations can find their operands at the top of the stack when they need them. Fortunately, if the target stack machine supports the necessary stack manipulation operations, then the stack can often be manipulated such that input operands are made available to the operation that consumes them, as needed. So long as the cost of such manipulation is less than the cost of the corresponding loads and stores from a local variable then caching a temporary on the stack can pay off.

Several efforts have been made studying the opportunities and effects of replacing loads and stores with stack manipulations. Koopman [1994] conducted a preliminary investigation of such optimization, laying out the basic process of searching for pairs of loads from a local variable in a basic block and preserving the value from the first load so the second load can be eliminated, referring to this as *stack scheduling*. The copied value was always sent to the bottom of the stack and brought up to the top when needed later. This transformation was to be used with other simple optimizations, such as those which eliminate variables killed by the transformation.

Maierhofer and Ertl [1998] expand on Koopman [1994] by noting that the copied value need not be pushed all the way to the bottom of the stack, but only out of the way of intervening instructions. They

make explicit the need for a live-variable analysis of the block before the transformation is done, so that pairs of the same definition of a variable, and not simply the same variable, are matched. They show how to compute optimal stack allocation using a dependence graph among the instructions; since optimal allocation is exponential, they give heuristics for a good stack allocation that is faster to compute. They also define a cost model by which they claim optimality, under which they assume that accesses to local variables cost three times as much as stack manipulations. They describe how their transformation must search backward through the code for loads of local variables.

Vallée-Rai et al. [2000] use an *ad hoc* approach in their Soot framework for optimizing Java bytecode. Class files are converted to a three-address intermediate representation, optimized using well-known techniques, and converted back to Java bytecode. Generating bytecode that makes effective use of the stack is an explicit goal for Soot, and the *ad hoc* techniques applied are similar to those originally implemented by Nystrom in our own BLOAT framework [Nystrom 1998; Whitlock 2000; Hosking et al. 2000].

Shpeisman and Tikir [1999] have listed specific instances for code replacement in Java bytecode. Their replacement strategies did simple analysis on code occurring between store and load and between two loads, and considered the many variations on `dup` bytecodes available for Java. They did not formalize the transformation or explicitly use the remaining Java stack manipulation operation, the `swap` bytecode.

	Source code	Before stack allocation	After stack allocation
A	<code>b = a * a;</code>	<code>iload Local\$1 iload Local\$1 imul istore Local\$2</code>	<code>iload Local\$1 dup imul istore Local\$2</code>
B	<code>b = (a + 5) / a;</code>	<code>iload Local\$1 ldc 5 iadd iload Local\$1 idiv istore Local\$4</code>	<code>iload Local\$1 dup ldc 5 iadd swap idiv istore Local\$4</code>
C	<code>a = 5; b = a + 6;</code>	<code>ldc 5 istore Local\$1 iload Local\$1 ldc 6 iadd istore Local\$2</code>	<code>ldc 5 dup istore Local\$1 ldc 6 iadd istore Local\$2</code>
D	<code>a = 5; b = 7; c = 6 - a;</code>	<code>ldc 5 istore Local\$1 ldc 7 istore Local\$2 ldc 6 iload Local\$1 isub istore Local\$3</code>	<code>ldc 5 dup istore Local\$1 ldc 7 istore Local\$2 ldc 6 swap isub istore Local\$3</code>

The table above shows four examples of the kind of stack allocation that is described in the above-mentioned papers. Example A gives a fragment of Java source code, its corresponding bytecode fragment as produced by Sun's `javac` compiler, and the result after stack allocation is applied to the bytecode fragment. The contents of local variable `Local$1` (representing the local variable `a` in the source code) is loaded (i.e., pushed) twice to the stack. The instruction `imul` pops these two operands, multiplies them, and pushes the result on the stack. Finally, that result is stored (i.e., popped) to `Local$2`. The two `iload Local$1`

operations in succession imply loading a value identical to the one that is already at the top of the stack. This redundant read from memory can be replaced by a `dup` instruction, as seen in the right column. Example B has two loads from `LOCAL$1` that are separated by two other instructions, and illustrates the effect of stack manipulation instructions to bring an operand to the correct stack position for it to be consumed. The instructions leave a value on the stack (the result of `iadd`); if the value loaded from `LOCAL$1` is duplicated after the first load, this value will be below the result of the intervening code when needed. Thus we need a `swap` instruction in place of the second `iload` to put the stack into a proper state.

Example C shows, in the unoptimized code, a store to variable `a` (represented by `LOCAL$1`) followed immediately by a load from the same variable. That is, a value is pushed back on the stack immediately after it has been popped. Duplicating the value before it is removed eliminates the need to reload it. Note also that if all the loads from a definition of a variable (or, when using SSA, all the uses of that variable) are eliminated, the definition is dead and the store can be removed. (More accurately, it would be replaced by a `pop`, since the value is on the stack and needs to be removed. A peephole optimizer can then remove the `dup/pop` pair, as the `pop` cancels the effect of the `dup`.) It seems realistic to hope for the elimination of all of a variable's uses in code that has been transformed by optimizations which create short-lived temporary variables. In example D, the code from an entire statement in the Java source comes between the store to and load from `LOCAL$1`. We can still preserve a value on the stack while the statement is computed. Again, this may enable the elimination of a store.

As these examples suggest, the transformations search for sequences containing a load or store of a given variable followed by a load of the same variable, perhaps with other instructions in between. Such intervening code must have certain properties. First, there can be no intervening store to the variable to invalidate the original definition, otherwise duplicating and using the original value would preserve an out-of-date value that is invalid at the latter use. Second, we must be able to place a duplicated value at a position in the stack such that the intervening code will neither disturb it, nor increase the height of the stack to put the duplicated value out of reach of its subsequent use. Missing from the prior work is a generic encoding and formalization of the specific patterns in the bytecode that can be replaced

2 Our Result

We present two program transformations that generalize the ones noticed in the papers cited above; we define a static analysis that enables the transformations, and we prove correctness, that is, the code before a transformation is semantically equivalent to the transformed code. We work with the following grammar for an idealized subset of JVM bytecodes: these include several bytecodes with their usual semantics, but we generalize `dup` and the `swap` variants with `dupxp` and `rollq`, respectively, with semantics as given below. In keeping with this semantics, where convenient we will refer to `dupx0` and `roll1` using their Java bytecode names `dup` and `swap`, respectively. We use ε to denote the empty string of bytecodes, and we denote concatenation of bytecode sequences by juxtaposition.

$$\begin{aligned}
e &\in \text{Instruction}^* \\
i &\in \text{Instruction} \\
i &::= \text{istore } v \mid \text{iload } v \mid \text{ldc } c \mid \text{iadd} \mid \text{dup}_{xp} \mid \text{roll}_q \\
v &\in \text{Var} \quad (\text{a set of variable names}) \\
c &\in \text{Int} \quad (\text{the set of integers}) \\
n, p, q &\in \text{Nat} \quad (\text{the set of non-negative integers})
\end{aligned}$$

We will define a small-step operational semantics using three semantic domains:

$$s, t \in \text{Stack} = \text{Int}^*$$

$$\begin{aligned}
R &\in \text{Store} = \text{Var} \mapsto \text{Int} \\
(e, s, R) &\in \text{State} = \text{Instruction}^* \times \text{Stack} \times \text{Store}.
\end{aligned}$$

A *stack* is a sequence of integers with the top on the left. We use `nil` to denote the empty stack, and we denote concatenation of stacks by \bullet (e.g., $s \bullet s'$). A *store* is a finite mapping of variables to integers. We use $\text{dom}(R)$ to denote the set of variables defined in R (i.e., the domain of R). A *state* is a triple consisting of an instruction, a stack, and a store. A *final state* is a state (e, s, R) where $e = \varepsilon$. We will use the abbreviation (s, R) for a final state (ε, s, R) .

The semantics is given by the reflexive, transitive closure of the following rules:

$$\begin{aligned}
(\text{istore } v \ e, s, R) &\rightarrow (e, s', R[v := c]) & s &= (c) \bullet s' \\
(\text{iload } v \ e, s, R) &\rightarrow (e, (R(v)) \bullet s, R) & v &\in \text{dom}(R) \\
(\text{ldc } c \ e, s, R) &\rightarrow (e, (c) \bullet s, R) \\
(\text{iadd } e, s, R) &\rightarrow (e, (c_1 + c_2) \bullet s', R) & s &= (c_1) \bullet (c_2) \bullet s' \\
(\text{dup_xp } e, s, R) &\rightarrow (e, (c_0) \bullet (c_1) \bullet \dots \bullet (c_p) \bullet (c_0) \bullet s', R) & s &= (c_0) \bullet (c_1) \bullet \dots \bullet (c_p) \bullet s' \\
(\text{roll_q } e, s, R) &\rightarrow (e, (c_q) \bullet (c_0) \bullet \dots \bullet (c_{q-1}) \bullet s', R) & s &= (c_0) \bullet \dots \bullet (c_{q-1}) \bullet (c_q) \bullet s'
\end{aligned}$$

It is straightforward to prove that if $((e_1 \ e_2), s, R) \rightarrow^* (e_2, s', R')$, then $((e_1 \ e_3), s, R) \rightarrow^* (e_3, s', R')$. We will use this observation repeatedly and without reference.

We use two functions to analyze instruction sequences: $\text{change}(e)$ calculates the total change in the stack height over the sequence of instructions e , and $\text{needs}(e)$ calculates the depth of the stack that e reads.

$$\begin{array}{ll}
\text{change}(\varepsilon) = 0 & \text{needs}(\varepsilon) = 0 \\
\text{change}(\text{istore } v \ e) = \text{change}(e) - 1 & \text{needs}(\text{istore } v \ e) = \max(1, \text{needs}(e) + 1) \\
\text{change}(\text{iload } v \ e) = \text{change}(e) + 1 & \text{needs}(\text{iload } v \ e) = \max(0, \text{needs}(e) - 1) \\
\text{change}(\text{ldc } c \ e) = \text{change}(e) + 1 & \text{needs}(\text{ldc } c \ e) = \max(0, \text{needs}(e) - 1) \\
\text{change}(\text{iadd } e) = \text{change}(e) - 1 & \text{needs}(\text{iadd } e) = \max(2, \text{needs}(e) + 1) \\
\text{change}(\text{dup_xp } e) = \text{change}(e) + 1 & \text{needs}(\text{dup_xp } e) = \max(p + 1, \text{needs}(e) - 1) \\
\text{change}(\text{roll_q } e) = \text{change}(e) & \text{needs}(\text{roll_q } e) = \max(q + 1, \text{needs}(e))
\end{array}$$

Notice that $\text{needs}(e) \geq 0$ for all e . Notice also that $\text{needs}(\text{istore } v \ e) = \text{needs}(e) + 1$ for all e . Notice finally that $\text{needs}(\text{dup}) = 1$ and $\text{needs}(\text{swap}) = 2$ because those instructions need that many values on the stack to execute. We consider an instruction to consume any values it reads, even if puts the same values back on the stack. We use $\text{height}(s)$ for the height of stack s (i.e., the number of items in the stack): $\text{height}(\text{nil}) = 0$, $\text{height}((c) \bullet s) = 1 + \text{height}(s)$.

Intuitively, a sequence of instructions pops certain elements from the stack (its operands), leaves others, and pushes more (its results) in place of what it took. When the stack reaches its low point during the computation, it has shrunk by $\text{needs}(e)$ values. The substack remaining then stays intact during the whole computation. The final change on the stack plus the size of the consumed substack is the size of the substack added by e , as expressed by the following lemma.

Lemma 1. [Stack size] *If $(e, s_1, R_1) \rightarrow^* (s_2, R_2)$, then there exist s_3, s_4, s_5 such that $s_1 = s_4 \bullet s_3$, $s_2 = s_5 \bullet s_3$, $\text{height}(s_4) = \text{needs}(e)$, and $\text{height}(s_5) = \text{change}(e) + \text{height}(s_4)$.*

Proof. We proceed by induction on the structure of e . If $e = \varepsilon$, then $s_1 = s_2$, so we can chose $s_3 = s_1$, $s_4 = \text{nil}$, and $s_5 = \text{nil}$. We have $\text{change}(e) = 0$ and $\text{needs}(e) = 0$, so $\text{height}(s_4) = \text{needs}(e)$ and $\text{height}(s_5) = \text{change}(e) + \text{height}(s_4)$. If $e = i \ e'$, then there are 6 cases of i .

If $i = \text{istore } v$, then $\text{change}(e) = \text{change}(e') - 1$, and hence $\text{change}(e') = \text{change}(e) + 1$. Moreover, $\text{needs}(e) = \text{needs}(e') + 1$. Hence $\text{needs}(e') = \text{needs}(e) - 1$. Then by induction, there exist s_6, s_7, s_8

such that $(\text{istore } v \ e', (c) \bullet s_7 \bullet s_6, R) \rightarrow (e', s_7 \bullet s_6, R[v := c]) \rightarrow^* (s_8 \bullet s_6, R')$ where $\text{height}(s_7) = \text{needs}(e') = \text{needs}(e) - 1$ and $\text{height}(s_8) = \text{change}(e') + \text{needs}(e') = \text{change}(e) + 1 + \text{needs}(e) - 1 = \text{change}(e) + \text{needs}(e)$. Set $s_3 = s_6$, $s_4 = (c) \bullet s_7$ and $s_5 = s_8$. Then $\text{height}(s_4) = 1 + \text{needs}(e) - 1 = \text{needs}(e)$ and $\text{height}(s_5) = \text{change}(e) + \text{needs}(e)$.

If $i = \text{iload } v$ or $\text{ldc } c$, then $\text{change}(e) = \text{change}(e') + 1$, and hence $\text{change}(e') = \text{change}(e) - 1$. Moreover, $\text{needs}(e) = \max(0, \text{needs}(e') - 1)$. There are two cases:

- If $\text{needs}(e) = 0$ and $\text{needs}(e') = 0$, then by induction there exist s_6, s_7, s_8 , namely $s_7 = \text{nil}$ and $s_6 = (c) \bullet s_9$ for some c and s_9 , such that $(i \ e', s_9, R) \rightarrow (e', s_7 \bullet s_6, R) \rightarrow^* (s_8 \bullet s_6, R')$ where $\text{height}(s_8) = \text{change}(e') = \text{change}(e) - 1$. Set $s_3 = s_9$, $s_4 = \text{nil}$ and $s_5 = s_8 \bullet (c)$. Then $\text{height}(s_4) = 0 = \text{needs}(e)$ and $\text{height}(s_5) = \text{height}(s_8) + 1 = \text{change}(e) - 1 + 1 = \text{change}(e) = \text{change}(e) + \text{needs}(e)$.
- If $\text{needs}(e') = \text{needs}(e) + 1$, then by induction there exist s_6, s_7, s_8 , namely $s_7 = (c) \bullet s_9$ for some s_9 , such that $(i \ e', s_9 \bullet s_6, R) \rightarrow (e', s_7 \bullet s_6, R) \rightarrow^* (s_8 \bullet s_6, R')$ where $\text{height}(s_7) = \text{needs}(e') = 1 + \text{needs}(e)$ and $\text{height}(s_8) = \text{change}(e') + \text{needs}(e') = \text{change}(e) - 1 + 1 + \text{needs}(e) = \text{change}(e) + \text{needs}(e)$. Set $s_3 = s_6$, $s_4 = s_9$, and $s_5 = s_8$. Then $\text{height}(s_4) = \text{height}(s_7) - 1 = 1 + \text{needs}(e) - 1 = \text{needs}(e)$ and $\text{height}(s_5) = \text{change}(e) + \text{needs}(e)$.

If $i = \text{add } v$, then $\text{change}(e) = \text{change}(e') - 1$, and hence $\text{change}(e') = \text{change}(e) + 1$. Moreover, $\text{needs}(e) = \max(2, \text{needs}(e') + 1)$. There are two cases:

- If $\text{needs}(e) = 2$ and $\text{needs}(e') = 0$, then by induction there exist s_6, s_7, s_8 , namely $s_7 = \text{nil}$, and $s_6 = (c_1 + c_2) \bullet s_9$ for some s_9, c_1, c_2 , such that $(\text{add } e', (c_1) \bullet (c_2) \bullet s_9, R) \rightarrow (e', s_7 \bullet s_6, R) \rightarrow^* (s_8 \bullet s_6, R')$ where $\text{height}(s_8) = \text{change}(e') + \text{needs}(e') = \text{change}(e) + 1$. Set $s_3 = s_9$, $s_4 = (c_1) \bullet (c_2)$ and $s_5 = s_8 \bullet (c_1 + c_2)$. Then $\text{height}(s_4) = 2 = \text{needs}(e)$ and $\text{height}(s_5) = \text{change}(e) + 1 + 1 = \text{change}(e) + \text{needs}(e)$.
- If $\text{needs}(e) \geq 2$ and $\text{needs}(e') = \text{needs}(e) - 1$, then by induction there exist s_6, s_7, s_8 , namely $s_7 = (c_1 + c_2) \bullet s_9$ for some s_9, c_1, c_2 , such that $(\text{add } e', (c_1) \bullet (c_2) \bullet s_9 \bullet s_6, R) \rightarrow (e', s_7 \bullet s_6, R) \rightarrow^* (s_8 \bullet s_6, R')$ where $\text{height}(s_7) = \text{needs}(e') = \text{needs}(e) - 1$ and $\text{height}(s_8) = \text{change}(e') + \text{needs}(e') = \text{change}(e) + 1 + \text{needs}(e) - 1 = \text{change}(e) + \text{needs}(e)$. Set $s_3 = s_6$, $s_4 = (c_1) \bullet (c_2) \bullet s_9$ and $s_5 = s_8$. Then $\text{height}(s_4) = 2 + \text{height}(s_9) = 1 + \text{height}(s_7) + 1 + \text{needs}(e) - 1 = \text{needs}(e)$ and $\text{height}(s_5) = \text{change}(e) + \text{needs}(e)$.

If $i = \text{dup_xp}$, then $\text{change}(e) = \text{change}(e') + 1$, and hence $\text{change}(e') = \text{change}(e) - 1$. Moreover, $\text{needs}(e) = \max(p + 1, \text{needs}(e') - 1)$. There are three cases:

- If $p + 1 > \text{needs}(e') = 0$ (and so $\text{needs}(e) = p + 1$) then by induction there exist s_6, s_7, s_8 , namely $s_7 = \text{nil}$, $s_6 = (c) \bullet s_9 \bullet (c) \bullet s_{10}$ for some c, s_9, s_{10} such that $(\text{dup_xp } e', (c) \bullet s_9 \bullet s_{10}, R) \rightarrow (e', s_7 \bullet s_6, R) \rightarrow^* (s_8 \bullet s_6, R')$ where $\text{height}(s_8) = \text{change}(e') + \text{needs}(e') = \text{change}(e) - 1$ and $\text{height}(s_9) = p$. Set $s_3 = s_{10}$, $s_4 = (c) \bullet s_9$ and $s_5 = s_8 \bullet (c) \bullet s_9 \bullet (c)$. Then $\text{height}(s_4) = 1 + p = \text{needs}(e)$ and $\text{height}(s_5) = \text{height}(s_8 \bullet (c) \bullet s_9 \bullet (c)) = \text{change}(e) - 1 + 1 + p + 1 = \text{change}(e) + p + 1 = \text{change}(e) + \text{needs}(e)$.
- If $p + 1 > \text{needs}(e') > 0$ (and so $\text{needs}(e) = p + 1$) then by induction there exist s_6, s_7, s_8 , namely $s_7 = (c) \bullet s_{11}$, $s_6 = s_9 \bullet (c) \bullet s_{10}$ for some c, s_9, s_{10}, s_{11} such that $(\text{dup_xp } e', (c) \bullet s_{11} \bullet s_9 \bullet s_{10}, R) \rightarrow (e', s_7 \bullet s_6, R) \rightarrow^* (s_8 \bullet s_6, R')$ where $\text{height}(s_8) = \text{change}(e') + \text{needs}(e') = \text{change}(e) - 1$ and $\text{height}(s_{11} \bullet s_9) = p$. Also $\text{height}((c) \bullet s_{11}) = \text{height}(s_7) = \text{needs}(e')$ so $\text{height}(s_{11}) = \text{needs}(e') - 1$. Hence $\text{height}(s_9) = \text{height}(s_{11} \bullet s_9) - \text{height}(s_{11}) = p - \text{needs}(e') + 1$. Set $s_3 = s_{10}$, $s_4 = (c) \bullet s_{11} \bullet s_9$ and $s_5 = s_8 \bullet s_9 \bullet (c)$. Then $\text{height}(s_4) = 1 + p = \text{needs}(e)$ and $\text{height}(s_5) = \text{height}(s_8 \bullet s_9 \bullet (c)) = \text{change}(e) - 1 + \text{needs}(e') + p - \text{needs}(e') + 1 + 1 = \text{change}(e) + p + 1 = \text{change}(e) + \text{needs}(e)$.

- If $p+1 \leq \text{needs}(e')$ and so $\text{needs}(e) = \text{needs}(e') - 1$ and $\text{needs}(e') = \text{needs}(e) + 1$, then by induction there exist s_6, s_7, s_8 , namely $s_7 = (c) \bullet s_{10} \bullet (c) \bullet s_9$, for some c, s_{10} such that $(\text{dup_xp } e', (c) \bullet s_{10} \bullet s_9 \bullet s_6, R) \rightarrow (e', s_7 \bullet s_6, R) \rightarrow^* (s_8 \bullet s_6, R')$ where $\text{height}(s_7) = \text{needs}(e') = 1 + \text{needs}(e)$ and $\text{height}(s_8) = \text{change}(e') + \text{needs}(e') = \text{change}(e) - 1 + \text{needs}(e) + 1 = \text{change}(e) + \text{needs}(e)$. Set $s_3 = s_6, s_4 = (c) \bullet s_{10} \bullet s_9$ and $s_5 = s_8$. Then $\text{height}(s_4) = \text{height}((c) \bullet s_{10} \bullet s_9) = \text{height}((c) \bullet s_{10} \bullet (c) \bullet s_9) - 1 = \text{height}(s_7) - 1 = 1 + \text{needs}(e) - 1 = \text{needs}(e)$ and $\text{height}(s_5) = \text{change}(e) + \text{needs}(e)$.

The case of $i = \text{roll_q}$ is similar to that of $i = \text{dup_xp}$, we omit the details. \square

The function $\text{needs}(e)$ determines how large a portion of the stack is used by an instruction sequence e . The following lemma says that altering the bottom of the stack not included in $\text{needs}(e)$ does not affect the change to the store across the computation of e .

Lemma 2. [Stack independence] *If $(e, s \bullet s_1, R) \rightarrow^* (s' \bullet s_1, R')$ and $\text{needs}(e) = \text{height}(s)$, then $(e, s \bullet s'_1, R) \rightarrow^* (s' \bullet s'_1, R')$*

Proof. We proceed by induction on the structure of e . If $e = \varepsilon$, then $s = s'$ and $R = R'$. If $e = i e'$, then there are 6 cases of i .

If $i = \text{istore } v$, then $\text{height}(s) = \text{needs}(e) = \text{needs}(i e') = 1 + \text{needs}(e') \geq 1$, so we can write $s = (c) \bullet s'$. We also have $\text{height}(s) = 1 + \text{height}(s')$, so $\text{needs}(e') = \text{height}(s')$. Now

$$(e, s \bullet s_1, R) = ((i e'), (c) \bullet s' \bullet s_1, R) \rightarrow (e', s' \bullet s_1, R[v := c]) \rightarrow^* (s' \bullet s_1, R')$$

so from the induction hypothesis we have

$$(e, s \bullet s'_1, R) = ((i e'), (c) \bullet s' \bullet s'_1, R) \rightarrow (e', s' \bullet s'_1, R[v := c]) \rightarrow^* (s' \bullet s'_1, R').$$

The other five cases are similar, we omit the details. \square

A transformation is correct if the code before and after the transformation is equivalent:

$$e_1 \equiv e_2 \text{ iff } [\text{for all } s, R, s', R' : (e_1, s, R) \rightarrow^* (s', R') \text{ iff } (e_2, s, R) \rightarrow^* (s', R')].$$

It is straightforward to show that \equiv is an equivalence relation. It is also straightforward to show that \equiv is a congruence, that is, if $e_1 \equiv e_2$, then $e_0 e_1 e_3 \equiv e_0 e_2 e_3$ for all e_0, e_3 .

Theorem 3. [Correctness] *Suppose $\text{istore } v$ does not occur in e .*

- If $\text{change}(e) = q - p - 1$ and $\text{needs}(e) = p + 1$,
then $\text{iload } v (\text{dup})^n e \text{ iload } v \equiv \text{iload } v (\text{dup})^n \text{dup_xp } e \text{ roll_q}$.*
- If $\text{change}(e) = q - p$ and $\text{needs}(e) = p$,
then $\text{istore } v e \text{ iload } v \equiv \text{dup_xp } \text{istore } v e \text{ roll_q}$.*

Proof. Consider first (a). We will show the direction from left to right: if we can execute the left-hand side as shown, then we can execute the right-hand side as shown.

$$\begin{array}{ll}
(\text{iload } v (\text{dup})^n e \text{ iload } v, t, R) & (\text{iload } v (\text{dup})^n \text{dup_xp } e \text{ roll_q}, t, R) \\
\rightarrow ((\text{dup})^n e \text{ iload } v, (R(v)) \bullet t, R) & \rightarrow ((\text{dup})^n \text{dup_xp } e \text{ roll_q}, (R(v)) \bullet t, R) \\
\rightarrow^n (e \text{ iload } v, (R(v))^{n+1} \bullet t, R) & \rightarrow^n (\text{dup_xp } e \text{ roll_q}, (R(v))^{n+1} \bullet t, R) \\
\rightarrow^* (\text{iload } v, t', R') & \rightarrow (e \text{ roll_q}, s'_4 \bullet (R(v)) \bullet s_3, R) \\
\rightarrow ((R'(v)) \bullet t', R') & \rightarrow^* (\text{roll_q}, s_5 \bullet (R(v)) \bullet s_3, R') \\
& \rightarrow ((R(v)) \bullet s_5 \bullet s_3, R').
\end{array}$$

In the execution of the left-hand side, notice that since we can execute the first occurrence of `iload v`, it must be the case that $v \in \text{dom}(R)$. Regarding t' , we have from Lemma 1 that we can choose s_3, s_4, s_5 such that $t = s_4 \bullet s_3$, $t' = s_5 \bullet s_3$, $\text{height}(s_4) = \text{needs}(e)$, and $\text{height}(s_5) = \text{change}(e) + \text{height}(s_4)$. We will now justify that the right-hand side can be executed as shown. The first step can be taken because we have established that $v \in \text{dom}(R)$. The next n steps can be taken because each `dup` instruction finds at least one element on the stack. The `dup_xp` instruction can be executed because $\text{height}((R(v))^{n+1} \bullet t) \geq \text{height}(t) \geq \text{height}(s_4) = \text{needs}(e) = p + 1$. We choose s'_4 such that $\text{height}(s'_4) = p + 1$ and $(R(v))^{n+1} \bullet t = s'_4 \bullet s_3$. From Lemma 2, $(R(v))^{n+1} \bullet t = s'_4 \bullet s_3$, and $\text{needs}(e) = p + 1 = \text{height}(s'_4)$ we have that e can be executed, that the resulting store is R' , and that the resulting stack is of the form $s_5 \bullet (R(v)) \bullet s_3$. Finally, `roll_q` can be executed because $\text{height}(s_5) = \text{change}(e) + \text{height}(s_4) = q - p - 1 + p + 1 = q$. We have $t' = s_5 \bullet s_3$. Finally, since `istore v` does not occur in e , we have $R(v) = R'(v)$. The direction from right to left can be proved in a similar fashion.

Next consider (b). We will show the direction from left to right: if we can execute the left-hand side as shown, then we can execute the right-hand side as shown.

$$\begin{array}{ll}
(\text{istore } v \ e \ \text{iload } v, s, R) & (\text{dup_xp } \text{istore } v \ e \ \text{roll_q}, s, R) \\
\rightarrow (e \ \text{iload } v, t, R[v := c]) & \rightarrow (\text{istore } v \ e \ \text{roll_q}, (c) \bullet s'_4 \bullet (c) \bullet s_3, R) \\
\rightarrow^* (\text{iload } v, t', R') & \rightarrow (e \ \text{roll_q}, s'_4 \bullet (c) \bullet s_3, R[v := c]) \\
\rightarrow ((c) \bullet t', R'). & \rightarrow^* (\text{roll_q}, s_5 \bullet (c) \bullet s_3, R') \\
& \rightarrow ((c) \bullet s_5 \bullet s_3, R').
\end{array}$$

In the execution of the left-hand side, notice that since we can execute the `istore v` instruction, it must be the case that t is nonempty, say, $s = (c) \bullet t$. Regarding t' , we have from Lemma 1 that we can choose s_3, s_4, s_5 such that $t = s_4 \bullet s_3$, $t' = s_5 \bullet s_3$, $\text{height}(s_4) = \text{needs}(e)$, and $\text{height}(s_5) = \text{change}(e) + \text{height}(s_4)$. Finally, since `istore v` does not occur in e , we have $R'(v) = (R[v := c])(v) = c$. We will now justify that the right-hand side can be executed as shown. The `dup_xp` instruction can be executed because $\text{height}(s) = \text{height}(t) + 1 \geq \text{height}(s_4) + 1 = \text{needs}(e) + 1 = p + 1$. We choose s'_4 such that $\text{height}(s'_4) = p$ and $t = s'_4 \bullet s_3$. The `istore v` instruction can be executed because it finds least one element on the stack. From Lemma 2, $t = s'_4 \bullet s_3$, and $\text{needs}(e) = p = \text{height}(s'_4)$ we have that e can be executed, that the resulting store is R' , and that the resulting stack is of the form $s_5 \bullet (c) \bullet s_3$. Finally, `roll_q` can be executed because $\text{height}(s_5) = \text{change}(e) + \text{height}(s_4) = q - p + p = q$. We have $t' = s_5 \bullet s_3$. The direction from right to left can be proved in a similar fashion. \square

Our calculus is not a true subset of Java bytecode because it uses generalized `dup_xp` and `roll_q` instructions. Java has `dup`, `dup_x1`, and `dup_x2`. The instruction `roll_0` is equivalent to doing nothing at all, and the Java `swap` is equivalent to `roll_1`. The transformations thus possible in Java bytecode are listed below, for an instruction sequence in which `istore v` does not occur.

<code>iload v (dup)ⁿ e iload v</code>	\equiv	<code>iload v (dup)ⁿ⁺¹ e</code>	$\text{change}(e) = -1$	$\text{needs}(e) = 1$
<code>istore v e iload v</code>	\equiv	<code>dup istore v e</code>	$\text{change}(e) = 0$	$\text{needs}(e) = 0$
<code>iload v (dup)ⁿ e iload v</code>	\equiv	<code>iload v (dup)ⁿ⁺¹ e swap</code>	$\text{change}(e) = 0$	$\text{needs}(e) = 1$
<code>istore v e iload v</code>	\equiv	<code>dup istore v e swap</code>	$\text{change}(e) = 1$	$\text{needs}(e) = 0$
<code>iload v (dup)ⁿ e iload v</code>	\equiv	<code>iload v (dup)ⁿ dup_x1 e</code>	$\text{change}(e) = -2$	$\text{needs}(e) = 2$
<code>istore v e iload v</code>	\equiv	<code>dup_x1 istore v e</code>	$\text{change}(e) = -1$	$\text{needs}(e) = 1$
<code>iload v (dup)ⁿ e iload v</code>	\equiv	<code>iload v (dup)ⁿ dup_x1 e swap</code>	$\text{change}(e) = -1$	$\text{needs}(e) = 2$
<code>istore v e load v</code>	\equiv	<code>dup_x1 istore v e swap</code>	$\text{change}(e) = 0$	$\text{needs}(e) = 1$
<code>iload v (dup)ⁿ e iload v</code>	\equiv	<code>iload v (dup)ⁿ dup_x2 e</code>	$\text{change}(e) = -3$	$\text{needs}(e) = 3$
<code>istore v e iload v</code>	\equiv	<code>dup_x2 istore v e</code>	$\text{change}(e) = -2$	$\text{needs}(e) = 2$
<code>iload v (dup)ⁿ e iload v</code>	\equiv	<code>iload v (dup)ⁿ dup_x2 e swap</code>	$\text{change}(e) = -2$	$\text{needs}(e) = 3$
<code>istore v e iload v</code>	\equiv	<code>dup_x2 istore v e swap</code>	$\text{change}(e) = -1$	$\text{needs}(e) = 2$

Acknowledgments. VanDrunen was supported by a Purdue Andrews Fellowship and an Intel Doctoral Fellowship. Hosking was supported in part by a National Science Foundation award, CCR-9711673, and by gifts from Sun Microsystems, Inc. Palsberg was supported in part by a National Science Foundation Faculty Early Career Development Award, CCR-9734265.

References

- BRIGGS, P., COOPER, K. D., HARVEY, T. J., AND SIMPSON, L. T. 1998. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience* 28, 8 (July), 859–881.
- BRIGGS, P., COOPER, K. D., AND TORCZON, L. 1994. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.* 16, 3 (May), 428–455.
- CHAITIN, G. J. 1982. Register allocation and spilling via graph coloring. In Proceedings of the ACM Symposium on Compiler Construction (Boston, Massachusetts, June). *ACM SIGPLAN Notices* 17, 6 (June), 98–105.
- CHOW, F., CHAN, S., KENNEDY, R., LIU, S.-M., LO, R., AND TU, P. 1997. A new algorithm for partial redundancy elimination based on SSA form. In Proceedings of the ACM Conference on Programming Language Design and Implementation (Las Vegas, Nevada, June). *ACM SIGPLAN Notices* 32, 5 (May), 273–286.
- CHOW, F. C. AND HENNESSY, J. L. 1990. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.* 12, 4 (Oct.), 501–536.
- CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the program dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct.), 451–490.
- ERTL, M. A. 1995. Stack caching for interpreters. In Proceedings of the ACM Conference on Programming Language Design and Implementation (La Jolla, California, June). *ACM SIGPLAN Notices* 30, 6 (June), 315–327.
- GEORGE, L. AND APPEL, A. W. 1996. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.* 18, 3 (May), 300–324.
- GERLEK, M. P., STOLTZ, E., AND WOLFE, M. 1995. Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. Program. Lang. Syst.* 17, 1 (Jan.), 85–122.
- HASTI, R. AND HORWITZ, S. 1998. Using static single assignment form to improve fbw-insensitive pointer analysis. See PLDI'98 [1998], 97–105.
- HOSKING, A. L., NYSTROM, N., WHITLOCK, D., CUTTS, Q., AND DIWAN, A. 2000. Partial redundancy elimination for access path expressions. *Software—Practice and Experience*. To appear in Special Issue on Aliasing in Object-Oriented Systems.
- KOOPMAN, P. J. 1994. A preliminary exploration of optimized stack code generation. *Journal of Forth Applications and Research*, 241–251.
- LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification*. Addison-Wesley.
- MAIERHOFER, M. AND ERTL, M. A. 1998. Local stack allocation. In *Proceedings of the International Conference on Compiler Construction* (Lisbon, Portugal, Mar.). Lecture Notes in Computer Science, vol. 1383. 189–203.
- MOREL, E. AND RENVOISE, C. 1979. Global optimization by suppression of partial redundancies. *Commun. ACM* 22, 2 (Feb.), 96–103.
- MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
- NYSTROM, N. J. 1998. Bytecode level analysis and optimization of Java classes. M.S. thesis, Purdue University.
- PLDI'98 1998. *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Montréal, Canada, June). Vol. 33.
- POLETO, M. AND SARKAR, V. 1999. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (Sept.), 895–913.
- SASTRY, A. V. S. AND JU, R. D. C. 1998. A new algorithm for scalar register promotion based on SSA form. See PLDI'98 [1998], 15–25.
- SHPEISMAN, T. AND TIKIR, M. 1999. Generating efficient stack code for Java. Tech. Rep. CS-TR-4069, University of Maryland. Oct.
- STOLZ, E., GERLEK, M. P., AND WOLFE, M. 1994. Extended SSA with factored use-def chains to support optimization and parallelism. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences* (Jan.). 43–52.
- VALLÉE-RAI, R., GAGNON, E., HENDREN, L. J., LAM, P., POMINVILLE, P., AND SUNDARESAN, V. 2000. Optimizing java bytecode using the Soot framework: Is it feasible? In *Proceedings of the International Conference on Compiler Construction* (Berlin, Germany, Apr.), D. A. Watt, Ed. Lecture Notes in Computer Science, vol. 1781. 18–34.
- WHITLOCK, D. 2000. Persistence-enabled optimization of java programs. M.S. thesis, Purdue University.
- WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley.