Transient variable caching in Java's stack-based intermediate representation

Paul Týma

Syracuse University, 26250 Euclid Ave. #503, Euclid,
OH 44132, USA

Tel.: +1 216 732 5895; Fax: +1 216 732 5897;

E-mail: ptyma@cat.syr.edu

Java's stack-based intermediate representation (IR) is typically coerced to execute on register-based architectures. Unoptimized compiled code dutifully replicates transient variable usage designated by the programmer and common optimization practices tend to introduce further usage (i.e., CSE, Loop-invariant Code Motion, etc.). On register based machines, often transient variables are cached within registers (when available) saving the expense of actually accessing memory. Unfortunately, in stack-based environments because of the need to push and pop the transient values, further performance improvement is possible. This paper presents Transient Variable Caching (TVC), a technique for eliminating transient variable overhead whenever possible. This optimization would find a likely home in optimizers attached to the back of popular Java compilers.¹ Side effects of the algorithm include significant instruction reordering and introduction of many stack-manipulation operations. This combination has proven to greatly impede the ability to decompile stack-based IR code sequences. The code that results from the transform is faster, smaller, and greatly impedes decompilation.

1. Introduction

It is a common operation to remove transient variables in register-based intermediate representations. This is likely done in contemporary just-in-time (JIT) compilers when Java is executed on register-based machines. However, many Java environments are interpreted and the usage of Java chips is nearing the horizon. These two environments will benefit directly from any instruction removal. The "cost" of all instructions

are relatively equivalent in interpreted environments and, in general, smaller code is faster code.

Java's stack-based instruction set is used as an intermediate representation for just-in-time and way-ahead-of-time [16] compilers. It is executed directly in many interpreted environments. The instruction set contains the canonical stack operations *push*, *pop*, *dup*, and *swap*. However, these instructions are relatively scarcely used in this form. Push operations are found in raw form with instructions such as *bipush* and *sipush* which push constant byte and short values respectively. More commonly, the use of *iload* instructions² retrieve values from a local variable (indexed within the *iload* instruction) and place it on the stack. The *getfield* instruction is its analogous counterpart for instance variables. Conversely, the *istore* and *putfield* instructions remove elements from the stack and save them to main storage. This paper will be primarily interested in this subset of the instruction set along with arithmetic instructions *iadd*, *isub*, etc.

This paper illustrates a technique to produce smaller, yet equivalent code sequences for common Java code. This is done through instruction reordering and peephole optimization often resulting in caching of transient variables on the stack. Stack-specific operations (e.g. *dup*) are used liberally to reduce code size

Preparation of the code includes standard basic block determination, control-flow and data-flow analysis. The literature covers these topics in depth [2] and they will not be considered here. We will assume the code segment in Fig. 1 is a basic block. We will also assume live-variable analysis has completed in this and all subsequent code segments identifying the variables as dead after the displayed usage.

Aside from input/output parameters, a function (method) should leave the stack in the same way it found it (i.e., after execution, the stack pointer points to the same position it did before execution). In effect, each

¹This optimization has been implemented by the author in the commercial Java bytecode optimizer "DashO-Pro".

²Instructions for dealing with floats, longs, doubles, etc. are also present. For consistency (and without loss of generality) we will only consider integer instructions.

```
// swapping A and B
int temp = A;   iload_1   iload_1
A = B;          istore_3   iload_2
B = temp;       iload_2   istore_1
                istore_1   istore_2
                iload_3
                istore_2
```

(a) Swap Code (b) Compiled (c) After TVC

Fig. 1. A basic block.

method is an input which *accepts* in a predefined push-down automata (PDA) with the functionality of each instruction being merely a side-effect. Typically, Java runtimes include bytecode verifiers designed to insure stack integrity. In this view, the primary interest of each instruction is whether it is a *push*, *pop*, or *delta*³ instruction.

At a more granular level, we can also see that each expression within basic blocks also must fulfill acceptance with the PDA. We may represent stack-based instruction sequences by a directed-acyclic graph (DAG) indicating the stack dependencies at each level. A basic block can be partitioned into a set of sub-blocks (represented by subtrees) to indicate sequences in the code which accept.

Each child of the basic block represents a sub-block which returns the stack to the state in which it found it. This will continue to be the prevailing requirement in creating and altering expression trees for stack-based code. The creation of the tree is subject to the attributes of each opcode. This affects the number of children and when a subtree is deemed accepting.

A delta instruction becomes a complete subtree. From our stack-oriented point-of-view they only exist because of their possible side effects (i.e., the function of the opcode). Typically, only those which use and/or define variables are of interest to TVC. Fig. 2a shows the DAG for the code segment in Fig. 1. The number notation associated with each edge signifies the stack state during a post-order traversal of the tree [5]. A post-order traversal represents the execution order of the code sequence. The first number representing the number of values cached on the stack as the traversal heads down the edge and the second number representing the number of values cached on the return (i.e., up the edge).

It is intuitive that adjacent definition-use code sequences for transient variables can be removed. We can

³A delta instruction is an instruction that leaves the stack unchanged.

remove these statements since a store immediately followed by a load simply pops and pushes the same value on the stack. Fig. 1 illustrates the opportunity in a simple example. Fig. 1a shows canonical code to swap two integer variables. In order to do so, we need temporary storage in the form of the *temp* variable. However, in consideration that the Java Virtual Machine is stack-based, we should just be able to use the stack itself as the needed temporary storage.

Fig. 1b shows the swap code compiled into the IR. Each variable is given a respective ordinal number (this is done by the compiler). The *temp* variable appears as variable 3. Fig. 1c shows the same code after the TVC algorithm. The store and load instructions for the temporary variable were analyzed and deemed unneeded. They were therefore removed. This example shows a crystal clear instance of the opportunity, significant code reordering is required to uncover opportunities in more complex code.

This type of opportunity presents itself often in complex expressions written by programmers or in code after common-subexpression-elimination (or other transforms) has been completed. The benefits of code removal are immediately apparent. In addition to eliminating the instruction decoding overhead, the removal of a store/load pair eliminates four memory accesses (assuming the stack is represented in memory as is typical in interpreted environments), and two stack pointer updates.

Fig. 2b illustrates the partitioning for the code sequence in Fig. 1 after the discussed optimization. When the store instruction was removed, the result of the computation remained on the stack until the following sub-block popped it in its *iadd* instruction.

It must be stressed that live-variable analysis of the entire scope of the suspected variable must be performed to assure it has no subsequent uses. We have avoided the unnecessary temp variable and we must be confident that it is not needed at a later point in the program. This idea is an initial hint at the goal of this work. The removal of these two statements allows the transient variable to remain cached on the top of the stack and save the needless execution of the store-load combination.

1.1. Algorithm

We are not limited to removing definition-use sequences only if they are adjacent. Definition-use sequences may always be removed as long as dependency relationships and stack integrity is preserved.

Fig. 2. Block organization.

Once a DAG is created from a basic block, definition-use sequences can be discovered and tested. In all cases, if we cache a value on top of the stack by removing a store, we must insure that the value will be available on top of the stack when the corresponding load would have been encountered. In general, Java compilers produce expressions in normal form [1]. That is, the program computes an expression tree and stores a result in memory. It proceeds to compute the next expression (possibly reloading the result of the first) and again stores a result and so on. From a performance standpoint, this introduces spurious store and load instructions.

The goal of TVC is to maximize the reduction of instructions due to the use of transient values. Subsequently, we will extend this goal to reduction of instructions for local variables with multiple uses. It is not always possible to remove all instances of transient variables. The algorithm is presented in several steps including a sequence of preparatory operations which increase the efficiency and execution of the general TVC algorithm.

1.2. Store-load scan

Each block is scanned from beginning to end and all store-load (i.e., definition-use) sequences are identified. During this initial scan, no special regard is given to the possibility of removal of these sequences. This runs in linear time for the number of instructions.

Definition 1. For some Procedure P:

Define $L(v)$ as the set of all load operations for variable v in P.

Define $S(v)$ as the set of all store operations for variable v in P where $L(v) \cap S(v) \equiv \emptyset$.

Define $R(v) \equiv L(v) \cup S(v)$ as the set of all references for variable v in P.

Define Γ_v^1 as the set of all definition-use pairs for some variable v in P such that $\forall \langle \gamma_s, \gamma_l \rangle \in \Gamma_v^1$ where $\gamma_s \in S(v)$ and $\gamma_l \in L(v)$, $\gamma_s < \gamma_l$ and there $\exists \rho \notin R(v)$ such that $(\rho > \gamma_s) \wedge (\rho < \gamma_l)$.

1.3. Liveness testing

The variable in question for each store-load pair is verified to be unused in subsequent code segments. It is assumed constant propagation and dead-code elimination have already taken place and removed instructions involved from our consideration. All possible execution paths must be tested to establish certainty that the load at the end of our store-load pair is the last use of this value. All pairs that prove to act on values which are live in subsequent blocks are removed from further consideration. Note that definition-use-definition sequences qualify as a dead value for the definition-use pair. We will see that we can also allow definition-use-use sequences within the same block to remain with some further reduction instruction removal. However, we must keep track as to not remove these sequences in our next preliminary step.

Definition 2. Define $\Gamma_v^2 \subseteq \Gamma_v^1$ where $\forall \langle \gamma_s, \gamma_l \rangle \in \Gamma_v^2$ that $\forall$ possible execution paths beyond γ_l the next $\rho \in R(v)$ encountered is a store, (i.e., $\in S(v)$) OR $\exists \rho \notin R(v)$ encountered for this path.

1.4. Removing adjacent pairs

An important criteria for the removal of a store-load pair is that the stack state at the time of the store is equivalent to the stack state at the time of the load. Simply, the stack pointer must be at the same position immediately after the store and immediately before the load. If this is not true, then the hasty removal of the

Fig. 3. Interlacing definition-use sequences.

pair will not have the temporary value on the top of the stack when the load would have pushed it there.

Fig. 3 exposes several properties of the instruction sequence. The dotted arcs on the left indicate whole subtrees. These are code segments which, upon completion, return the stack to the state in which they found it. The solid arcs on the right indicate matching definition-use pairs. Even when stack integrity would be maintained, removal of some pairs would destroy stack integrity for others. The removal of the store-load combination for arcs *A* and *B* are separately legal. However, removal of *A* or *B* destroys the stack state equilibrium for the other. This effect is easily extrapolated such that one removal can destroy many other possible removals. A general rule for the ability to remove a store-load pair is that the load is a leftmost child of its subtree (and the store roots the immediately preceding subtree).

An intuitive notion would be to determine how to remove the *most* pairs. Finding the ordering which yields the most removable pairs is NP-complete (reducing to longest transitive subgraph). Fortunately, the algorithm presented here attempts to use aggressive instruction reordering to bypass the problem.

Without issue, we could remove any pairs which not only have valid stack states, but their removal would not destroy the possibility of any other removals (i.e., *A* and *B* are in violation). This set is effectively pairs which are wholly contained or wholly disjoint from all other pairs. For purposes that will become clear, we will iterate a peephole optimizer to only remove pairs that are adjacent (that have no further use). This is clearly safe. In fact, the goal of the algorithm will be instruction reordering such that legal pairs become adjacent. The ability to reorder a pair to be adjacent proves the legality of its removal.

In Fig. 3, we only remove arc *E*. Arc *C* is stack imbalanced, arcs *A* and *B* are interlaced, and arc *D* is both. Thus we are left with definition-use pairs that do not have legal stack states or they are in conflict with other pairs.

1.5. Resolving violating pairs

As discussed, some pairs may not be removed because of the imbalance of the stack state between the store and the load as in arc *C* in Fig. 3.

Another problem is that of interlacing pairs. The removal of one of a set of interlaced pairs may destroy the ability to remove the other (arcs *A* and *B*). We will term both kinds of these problematic pairs (stack imbalanced and interlacing) as *violating* pairs.

In order to resolve violating pairs, we can reexamine the DAG for the given code sequence (see Fig. 4). Stack imbalances become immediately apparent with the help of the state indication on each edge.

In order to resolve these violations we can reorder the instructions in an attempt to make all pairs stack-safe and that each arc is wholly contained within one or more arcs or is disjoint with respect to other arcs.

1.5.1. Push migration

An instruction sequence may have many legal reorderings which retain the integrity of the original sequence [1]. We will only consider a subset of Aho's work and attempt to migrate push instructions towards the front of the block.

There are several rules to assure a legal move:

A push (load) instruction may move to an earlier position in a basic block so long as:

1. The instruction does not pass any other instruction s where $s \in S(v)$. Movement of a load instruction to a position before a store violates dependency requirements.

2. The instruction can only be repositioned ahead of an instruction which has the same stack state as it-

Fig. 4. DAG representation.

self (in the initial graph). Grafting an instruction to a place with a different stack state will cause its appearance at the top to occur at the wrong time. This requirement can be stated as: Only instructions which are (currently) leftmost children a subtree may be moved and they may not be inserted within existing subtrees. When a non-leftmost child instruction would beneficially be moved, if possible we will reorder the instructions to allow it to be a leftmost child.

3. Regardless of the distance between them, instructions which belong to the same subtree must retain their ordering. Any reordering of expression operands would destroy non-commutative operations. This can also be written that a push instruction may not migrate past an instruction with a stack state less than its own. Note that this requirement could be (and will be) relaxed with regard to commutative operators. However, in this initial presentation we will honor it in full force.

Formally:

Definition 3. Define B as a set of ordered statements $S_1 \dots S_n$ that form a basic block.

Define ϕ_x to be the stack state prior to position $S_x \in B$.

As earlier defined, $L(v) \subseteq R(v)$ and $S(v) \subseteq R(v)$ where $R(v) \subseteq B$.

Moving a statement S_i to before statement S_j where $i > j$ causes: $\forall S_x \in B$ such that $(x \geq j) \wedge (x < i)$, $S_x = S_{x+1}$ and $\phi_x = \phi_{x+1}$ and, $S_i = S_j$.

The move of a load instruction may not violate the meaning of the block where the requirements are:

A. The instruction which uses (pops) a loaded value must remain constant.

B. The depth at which the value is popped must remain constant.

C. The output of the code block must remain constant (dependent on A and B).

Theorem 1. A statement $S_i \in L(v)$ may be moved to before some statement S_j where $j < i$ without altering the meaning of the block iff:

1. $\exists S_x \notin S(v)$ such that $(x < i) \wedge (x > j)$.
2. $\phi_i = \phi_j$.
3. $\exists S_x \notin B$ such that $(x < i) \wedge (x > j)$ and $\phi_x < \phi_i$.

Proof. ($\Rightarrow$) Assume S_i is moved immediately ahead of S_j without altering the meaning of the block.

1. In consideration that $S_i \in L(v)$, any movement of this instruction ahead of some instruction $S_x \in S(v)$ for some given B violates dependency restrictions of the block. This move would alter the meaning of the block.

2. We can characterize the code sequence prior to migration as:

$$S_j, T_1 \dots T_n, S_i \dots POP_{S_i}$$

Where $S_z, T_z \in B$ and POP_{S_i} is the instruction that uses (pops) S_i . If $\phi_i = \phi_j$ then the sum of the stack movements for $T_1 \dots T_n$ is zero. If $(\phi_i > \phi_j) \vee (\phi_i < \phi_j)$ then the sum of the stack movements for $T_1 \dots T_n$ is positive or negative, respectively. Migrating ϕ_i prior to ϕ_j will cause S_i to be at a different stack depth at the time of its original position. Therefore, will violate requirements A and/or B for meaning of the block.

3. Assume that $\phi_i = \phi_j$, however also assume the contrary to requirement 3 that $\exists S_x \in B$ such that $(x < i) \wedge (x > j)$ and $\phi_x < \phi_i$. The sequence (prior to migration) can be characterized as:

$$S_j, T_1 \dots T_n, S_x, U_1 \dots U_n, S_i \dots POP_{S_i}$$

We will assume without loss of generality that S_x has a stack state of $\phi_i - 1$ and that S_x is the only instruction between S_j and S_i with a stack state less than ϕ_i . Since, by requirement 2 that $\phi_i = \phi_j$ the stack states for the above may be characterized with respect to ϕ_i as:

$$\phi_i, \phi_{T_1} \dots \phi_{T_n}, \phi_i - 1, \phi_{U_1} \dots \phi_{U_n}, \phi_i$$

Since the minimum stack state of the sequence is $\phi_i - 1$, we know that $\phi_{T_1} = \phi_{U_n} = \phi_i + 1$. After the migration, the sequences will appear as:

$$S_i, S_j, T_1 \dots T_n, S_x, U_1 \dots U_n, \dots POP_{S_i} \\ \phi_i, \phi_i + 1, \phi_{T_1} + 1 \dots \phi_{T_n} + 1, \phi_i, \phi_{U_1} + 1 \dots \phi_{U_n} + 1$$

The stack state of the migrated load S_i and S_x are equivalent. Therefore, whereas before the migration, the stack immediately prior to $\dots POP_{S_i}$ was:

$$\phi_x, \phi_i, \uparrow$$

it would be after the migration

$$\phi_i, \phi_x, \uparrow$$

This violates the meaning of the block by condition B.

($\Leftarrow$) Assume conditions 1–3 in the theorem. Following condition 1, we may eliminate as possible move-to targets all those ahead of the closest preceding store instruction. Following condition 2, we may eliminate all positions where the stack state is not equivalent to the load instruction we wish to move. Following condition 3, we eliminate all positions ahead of a lower stack state than that of instruction to be moved. In effect, this prevents us from changing the push ordering within any given subtree. Therefore, all positions left are dependency-safe, stack-safe, and output-consistent.

There is not always an apparent advantage in moving an instruction (i.e., a load with no preceding store, or any constant value push instruction). However, very often movement of an instruction might create oppor-

tunity for other instructions. With this in mind, we attempt to accomplish all legal moves. This typically creates a code sequence with a great deal of pushing at the beginning and popping at the end. This technique is well suited to improving interpreter performance. However, needless instruction movements could complicate static register allocation algorithms for Just-in-time compiled environments.

According to the movement requirements, our algorithm starts with the leftmost child of the rightmost subtree and attempt to migrate it forward. Note that starting with the rightmost child would work however, movement of the rightmost child would immediately fail requirement 3 as it tries to pass the leftmost child. We will soon remedy the inability of the rightmost child's movement.

In every migration we attempt to move the instruction as far as possible. Even though some cases allow us to move up an instruction to an intermediate level, we can reduce iterations of this algorithm by always moving instructions up as far as possible. The first position of the code sequence will always have a stack state of zero making it a prime target for early grafts. The load from pair *E* from Fig. 3 cannot move forward at all since it may not pass a store for the same variable (requirement 1). Consequently, the *bipush100* may also not move further since it may not pass the load in total ordering (requirement 3). Relaxation of requirement 3 for commutative operations is legal and will be discussed. The *iload3* of subtree 5 in Fig. 4 may move up to a position immediately succeeding the *istore3* in subtree 2. This move would cause the stack states of all instructions that were passed (subtrees 3 and 4) to be increased by one.

Fig. 5a shows the result after all possible moves have taken place. The dotted arrows on the left illustrate the new configuration of subtrees. Because of the the restrictions on the moves, none of these are interlaced. In fact, a test for a legal reordering of the instructions in a basic block is that all arcs are noninterlacing. The solid lines on the right illustrate that all pairs are now either disjoint or wholly contained. The push-migration procedure resolved the stack imbalance in pair *C* by forcing the *bipush55* to before the calculation of variable 3.

At this point, we can once again remove all adjacent pairs and are left with Fig. 5b. Pair *A* was not adjacent and was not removed. Pair *A* also is now stack imbalanced. When all adjacent pairs are removed, their subtree connections are grafted making larger subtrees. As can be seen in Fig. 5b, we are left with two sub-

Fig. 5. Iterations of push migration.

trees. Another iteration of push-migration can now legally move the *bipush55* and resolve the stack imbalance, putting pair A adjacent. The removal of A leaves Fig. 5c.

1.6. Commutative operations

Requirement 3 insures integrity of expressions throughout the algorithm. However, operands of commutative operations may be able to move regardless of the requirement. An operand of a commutative operations may move past another within the same subtree (effectively altering both instructions stack states) so long as they share the same parent and that parent is a commutative operation. Therefore, we allow restructuring an expression such as $x + y$ as $y + x$, while disallowing such reorderings for non-commutative operations (i.e., subtraction, division, etc.). This addition does not significantly change the paradigm and often exposes new opportunities for instruction removal. Commutative reordering may be beneficial even after two instructions have swapped positions several times (because of recent instruction removal, etc.). However, implementation requires careful consideration as to limit unconstrained instruction “leapfrogging”.

1.7. N-used definitions

Application of the TVC algorithm brings to light additional opportunities for peephole optimization. Often, this was a consequence of the natural tendency of the algorithm to group like instructions together. From there, these instructions could often be reduced to stack manipulation operations.

Instances of a definition-use-use sequence require special consideration. This may be applied to a definition followed by any number of uses. After push migration, an additional transform can be done to reduce the sequence as much as possible.

The code in Fig. 6a contains a store for variable 3 followed by two uses. For now, we will assume that variable 3 is dead after this code segment. We can still remove the definition-use sequence, as long as we replace the second load with a *dup*. Of course, this will reproduce the top-of-stack, correctly placing another instance at the correct time. Any number of contiguous loads could be replaced some combination of *dup* or *dup2* (duplicates the top two stack values) instructions. It is unfortunate no instruction exists to duplicate the top-of-stack x number of times. Heavy array usage

iload_1	iload_1	iload_1
iload_2	iload_2	iload_2
iadd	iadd	iadd
istore_3	istore_3	dup
iload_3	iload_3	bipush 100
bipush 100	iload_3	imul
imul	bipush 100	iadd
istore_1	imul	istore_2
iload_3	istore_1	
iload_1	iload_1	
iadd	iadd	
istore_2	istore_2	

(a) Original (b) Migration (c) Removal

Fig. 6. Multiple uses.

of Java produces significant numbers of load instructions as leftmost children. Currently, these degrade into a *dup* followed by numerous *dup2*'s.

In addition, regardless of liveness, any adjacent definition-use pair could be replaced with:

```
dup
istore x
```

The success of such a replacement is largely dependent upon the runtime implementation of true stack operations. If *dup* was consistently faster than a load, than this replacement is beneficial. For real stack architectures, this very well may be the case. However, for runtime systems placed on top of register architectures, it could very well be a dubious proposition to presume that true stack operations are thoughtfully implemented. On the other hand, runtime environments with knowledge of patterns produced by TVC (i.e., long sequences of DUPs) could optimize these instruction sequences. For example, a sequence of DUPs could be collapsed into a architecture-specific memory move. This type of cooperation between TVC and the runtime would exemplify performance improvement.

2. Application

Java's nature invoked certain restrictions upon the application of the TVC algorithm. In its described form, the algorithm could not be extended to be used upon instance variables. Since all *putfield* instructions could be visible to many threads, they could not be removed. In addition, safety dictates that we must assume all method calls change all instance variables.

Polymorphic method calls prevent us from insuring any given call is safe. A scheme is being devised to reduce some sets of sequential *getfield* operations to simple stack manipulations. In this scenario, thread safety is somewhat questionable – i.e., a given race condition could be transformed to a “different” race condition.

2.1. Effect on interpreters and just-in-time compilers

As has been said, any compression of size of the code is nearly linearly translated into performance improvement in interpreted environments. This is intuitive since the fetch-execute cycle of interpreted environments is linear and the decoding of each instruction is lost performance. Also, the concept of “expensive” instructions is diminished in interpreted environments.

Performance improvement in just-in-time compiled environments is less predictable. So far, JIT environments have been implemented in relatively different ways. Intuitively, the most prolific scheme would be based on register-caching schemes such as [7]. Such a paradigm may have difficulty working with resultant code of TVC. This is primarily due to the removal of named variable accesses and the introduction of stack-specific operations (i.e., *DUP*). In general, JIT environments benefit analogously to interpreted environments. Fewer instructions still mean less work to be done and this mantra still holds true into JIT platforms.

2.2. TVC for code obfuscation

Of significant interest has been the problem of the decompilation of Java programs. Commercial developers are haunted by the fact that their product's source code could be recreated using a decompilation program. Current “obfuscators” simply modify supporting data in class files in an effort to confuse decompilers, however they cannot change the bytecode itself. Consequently, “de-obfuscators” are relatively trivial to write. The bytecode can be extracted from an obfuscated class file and wrapped in fresh data structures once again allowing it to be decompiled.

Historically, performance transforms have hindered efforts for decompilation. TVC rings true in this effort. TVC's propensity at code reordering creates code sequences that cannot exist in Java source code. Fig. 1 is a perfect example. Fig. 1b can plainly be decompiled into three sequential load/store statements. However,

after TVC, Fig. 1c is meaningless when converted back to Java. Inherently this is because of TVC's ability to directly make use of the stack whereas the Java language provides no such constructs.

TVC's abilities for code obfuscation become more powerful as code length increases. N-used sequences destroy direct relations between pushes and pops. This feature proves to be a significant result of TVC.

3. Empirical results

As with many transforms, TVC typically provides a modest performance increase. However, unlike some transforms (i.e., code motion) TVC cannot increase execution time. This is guaranteed in interpreters. A JIT environment which makes significant assumptions about the incoming code could feasibly suffer, but this has yet to be observed.

Typical programs do not display such pristine opportunities as in Fig. 5. Typical programs processed through TVC did exhibit significant opportunity for instruction reordering, modest instruction removal (with corresponding code size reduction), and definitive impedance of decompilation. Results are presented here in counts of instruction reductions. All test cases were decompiled before and after TVC. All class files decompiled correctly before TVC but were significantly distorted after TVC. In many cases, decompilers were only able to disassemble the code. In others, small code sequences that TVC could not process were decompiled to a readable level. Results of all test programs were verified for correctness before and after (see Table 1).

CaffeineMark is a benchmark suite developed by Pendragon Software, Inc.

Results varied from no change up to 18% for different methods. This result is rather indicative of transforms such as peephole optimization.

4. Conclusion and future work

Early work in TVC has shown reduction in code segments beyond simple prune/grafting of expression trees. Primarily this is a testament to Java compilers' straightforward code generation. We have shown a technique for the removal of unnecessary instructions through instruction reordering. We have shown the criteria for "safe" reordering. This reordering made effective use of stack architecture while simultaneously impeding decompilation. As with optimizations such as CSE or peephole optimization, this algorithm would find a home within a bytecode optimizer. Its general ability being an enhancement to peephole optimization.

The next step is to extend the algorithm to a global generalization and relate it to specific syntactic constructs. Koopman [13] notes this technique can reduce loop overhead. I am currently working on a formalization of applying this technique to loops. In many cases, the loop control variable (which is often inherently transient) need only exist on the stack. Unrolling of loops immediately makes apparent the opportunity.

TVC is being tested as a global transform and its application to non-local variables. Stemming from this work, several other stack-based performance transforms are under analysis. TVC's generates complex code-sequence that are extremely difficult to verify. From this, work on a stack-state verification technique has ensued and is currently being formalized. Further stack-based transforms specifically to facilitate performance and/or obfuscation are being investigated.

5. Related work

Koopman [13] discusses a similar procedure for the removal of redundant local variable accesses. His work considers a mature stack environment utilizing operations outside the availability of the Java Virtual

Table 1

Class	Method	Original Size	After TVC	Reduction
CaffeineMarkFrame	handleEvent	371	359	3%
CaffeineMarkFrame	getGif	122	121	0%
java.util.Vector	trimToSize	44	36	18%
java.util.Properties	load	739	730	1%
sun.tools.javac.Main	getText	105	86	18%
sun.tools.javac.Main	compile	1983	1928	2%

Machine. Unlike the algorithm presented here, his algorithm favors the introduction of stack manipulation instructions instead of instruction reordering based on dependency analysis. Koopman also considers the problem by attacking def/use pairs in a smallest distance (between the def and the use) first sequence. His work was admittedly preliminary and only provided a hint at the effects of removal of redundant local variable accesses at a global level.

Acknowledgments

The author would like to thank Doug Lea, Garth Foster, and Nataraj Nagaratnam for numerous discussions and many suggestions for improving this paper.

References

- [1] A.V. Aho and S.C. Johnson, Optimal code generation for expression trees, *J. ACM* **23**(3) (July 1976).
- [2] A.V. Aho, R. Sethi and J. Ullman, *Compiler – Principles, Techniques, and Tools*, Addison-Wesley, 1988.
- [3] Y. Andoh, j2c: Java to C translator, www.webcity.co.jp/info/andoh/java/j2c.html, 1996.
- [4] H.E. Bal, The design and implementation of the EM global optimizer, Rapport IR-99, Vrije Universiteit, Amsterdam, March 1985.
- [5] J.L. Bruno and T. Lassagne, The generation of optimal code for stack machines, *J. ACM* **22**(3) (July 1975), pp. 382–396.
- [6] M. Anton Ertl, Stack caching for interpreters, in: *Proceedings of the ACM: Sigplan '95 Conference on PLDI*, June 1995, pp. 315–327.
- [7] M. Anton Ertl, Implementation of stack-based languages on register machines, April 1996, Dissertation University of Wien.
- [8] J. Gosling, Java intermediate bytecodes, in: *ACM SIGPLAN Workshop on Intermediate Representations*, 1995.
- [9] C.A. Hsieh, J. Gyllenhaal and W. Hwu, Java bytecode to native code translation: The Caffeine prototype and preliminary results, in: *IEEE Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996.
- [10] J.L. Keedy, On the evaluation of expressions using accumulators, stacks and store-to-store instructions, *Computer Architecture News* **7**(4) (1978), 24–27.
- [11] J.L. Keedy, More on the use of stacks in the evaluation of expressions, *Computer Architecture News* **7**(8) (June 1979), 18–21.
- [12] P. Koopman, Why stack machines?, *Usenet Nuggets*, *Computer Architecture News* **21**(1) (March 1993).
- [13] P. Koopman, A preliminary exploration of optimized stack code generation, in: *Proceedings of the 1992 Rochester Fort H Conference*, 1992.
- [14] P. Koopman, *Stack Computers: The New Wave*, Ellis Horwood Publishers, 1989.
- [15] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1997.
- [16] T.A. Proebsting et al., Toba: Java for applications a Way Ahead of Time (WAT) Compiler, <http://www.cs.arizona.edu/sumatra/toba>
- [17] Sun Microsystems, *picojava I Microprocessor Core Architecture*, <http://www.sun.com/sparc/whitepapers/wpr-0014-01/>
- [18] C.F. Schofield, *Optimizing Fortran Programs*, Ellis Horwood, Chichester, UK, 1989.
- [19] A.S. Tanenbaum, H. van Staveren and J.W. Stevenson, Using peephole optimization on intermediate code, *TOPLAS* **4**(1) (January 1982), 21–36.
- [20] A.S. Tanenbaum, H. van Staveren, E.G. Keizer, and J.W. Stevenson, A practical toolkit for making portable compilers, *CACM* **26**(9) (September 1983), 654–660.
- [21] A.S. Tanenbaum et al., Amsterdam Compiler Kit documentation, Rapport nr IR-90, Vrije Universiteit, Amsterdam, June 1984.
- [22] H.E. Bal and A.S. Tanenbaum, Language- and machine-independent global optimization on intermediate code, *Computer Languages* **11**(2) (April 1986), 105–121.
- [23] P. Týma, G. Torok and T. Downing, *Java Primer Plus*, Waite Group Press, 1996.
- [24] P. Týma, Tuning Java performance, *Dr. Dobbs Journal* (April 1996).
- [25] A. Wilk and W. Silverman, OPTIMA – A portable PCODE optimizer, *Software - Practice and Experience* **13**(4) (1983), 323–354.
- [26] N. Wirth, Stack vs. multiregister computers, *ACM Sigplan Notes* (March 1968), 13–19.