SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; **34**:1413–1439
Published online 25 August 2004 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.618

Anticipation-based partial redundancy elimination for static single assignment form

Thomas VanDrunen[1,*,†] and Antony L. Hosking[2]

[1] *Department of Mathematics and Computer Science, Wheaton College, 501 College Avenue, Wheaton, IL 60187-5593, U.S.A.*
[2] *Department of Computer Sciences, Purdue University, 250 N University St., West Lafayette, IN 47907-2055, U.S.A.*

SUMMARY

Partial redundancy elimination (PRE) is a program transformation that identifies and eliminates expressions that are redundant on at least one (but not necessarily all) execution paths of a program without increasing any path length. Chow, Kennedy and co-workers devised an algorithm (SSAPRE) for performing partial redundancy elimination on intermediate representations in static single assignment (SSA) form. The practicality of that algorithm is limited by the following concerns: (1) it makes assumptions about the namespace that are stronger than those of SSA form and that may not be valid if other optimizations have already been performed on the program; (2) if redundancies occur in nested expressions, the algorithm may expose but not eliminate them (requiring a second pass of the algorithm); (3) it misses cases covered by the state of the art in PRE; and (4) it is difficult to understand and implement. We present an algorithm (A-SSAPRE) structurally similar to SSAPRE that uses anticipation rather than availability; this algorithm is simpler than SSAPRE, covers more cases, eliminates nested redundancies on a single pass, and makes no assumptions about the namespace. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: partial redundancy elimination; static single assignment form; machine-independent program transformations

INTRODUCTION

Partial redundancy elimination (PRE) is a machine-independent optimizing program transformation, pioneered by Morel and Renvoise [1]. PRE identifies computations that may have been performed at an earlier point in the program and replaces such computations with reloads from temporaries. We say '*may* have been performed' because PRE considers computations that were performed on some, but not necessarily all, control flows paths to that point—in other words, computations that

*Correspondence to: Thomas VanDrunen, Department of Mathematics and Computer Science, Wheaton College, 501 College Avenue, Wheaton, IL 60187-5593, U.S.A.
†E-mail: Thomas.VanDrunen@wheaton.edu

Received 21 July 2003
Revised 29 March 2004
Accepted 30 March 2004

are *partially* redundant. In such cases, a computation must be inserted along paths to that program point that do not already compute that value, and care must be taken so that the length of no path is ever increased and no execution of the optimized program performs a computation that an execution of the unoptimized program does not. The net effect is to hoist code to earlier program points. This transformation subsumes common subexpression elimination and loop-invariant code motion.

OUR CONTRIBUTION

In this paper, we present a new algorithm for PRE on intermediate representations in static single assignment (SSA) form. An earlier SSA-based algorithm, SSAPRE, by Chow, Kennedy and co-workers [2,3], was weak in several ways. First, it made stronger assumptions about the form of the program (particularly the live ranges of variables) than are true for the traditional definition of SSA. Other SSA-based optimizations that are considered to preserve SSA may break these assumptions; if these other optimizations are performed before SSAPRE, the result may be incorrect. Moreover, the SSAPRE algorithm did not handle nested expressions and missed optimization opportunities that are covered by more recent redundancy-elimination algorithms [4,5]. Finally, from a software engineering perspective, the earlier PRE algorithm for SSA was difficult to understand and implement.

In this paper, after discussing preliminaries about the representation of the program to be optimized, we briefly describe SSAPRE and illustrate its shortcomings. Next we present our main contribution: a new algorithm for PRE that assumes and preserves SSA, called A-SSAPRE. It is structurally similar to, though simpler than, SSAPRE. The key difference is that it discovers redundancy by searching backwards from later computations that can be eliminated to earlier computations, rather than searching forward from early computations to later. Our running example will demonstrate that our new algorithm addresses the concerns about SSAPRE. Finally, we present performance results for A-SSAPRE and comment on directions for future work.

PRELIMINARIES

Throughout this paper, we will discuss transformations as they would be performed on an intermediate representation (IR) that uses a *control flow graph* (CFG) over *basic blocks*. Using standard terminology [6], a basic block is a code segment that has no unconditional jump or conditional branch statements except for possibly the last statement, and none of its statements, except possibly the first, is a target of any jump or branch statement. A CFG is a graph representation of a procedure that has basic blocks for nodes and whose edges represent the possible execution paths determined by jump and branch statements. In this paper, blocks are identified by numbers; paths in the graph (including edges) are identified by a parenthesized list of block numbers. We assume that all *critical edges*—edges from blocks with more than one successor to blocks with more than one predecessor [7]—have been removed from the CFG. Such an edge can be eliminated by inserting an empty block between the two blocks connected by the critical edge. We further assume that the instructions of the intermediate representation that interest us are in three-address form, saving their result to a variable or temporary. Another useful abstraction for control flow is the *dominator tree*. A node *dominates* another node if all paths to the second node must pass through the first. The nodes in the dominator tree are the same as

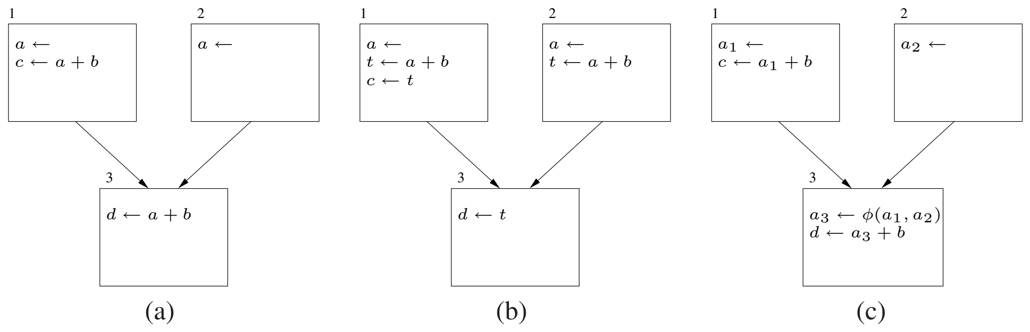

Figure 1. Basic example (a) before PRE; (b) after PRE; (c) in SSA form.

the nodes of the CFG, but the children of a given node are the nodes that are immediately dominated by that node; that is, nodes that are dominated by it and are not dominated by any other node also dominated by it (note that this excludes the node itself). The *dominance frontier* of a basic block is the set of blocks that are not dominated by that block but have a predecessor that is [8].

Having described the basics of program representation, we now consider an example of PRE performed on a code fragment in that representation. In the unoptimized version of this fragment in Figure 1(a), the expression $a+b$ is redundant when the control flow path (1,3) is taken, being computed twice without any change to the operands. However, the same computation is not redundant if the control flow path (2,3) is taken; even if it were computed some time earlier in the program, block 2 modifies one of the operands. Thus the expression is *partially redundant* in block 3. To remove the partially redundant computation, we insert a copy of it into block 2, since the value of the computation is unavailable along the incoming edge from it. This insertion makes the expression in block 3 fully redundant. By allocating a fresh temporary t and saving each early computation to it, the later computation can be replaced by that temporary, producing the optimized code found in Figure 1(b).

Notice that PRE does not reduce code size. Both the optimized and unoptimized versions in this example contain two occurrences of the computation. Moving the code, however, decreases the length of the left execution path. The length of the right execution path is unchanged.

Static single assignment form

Static single assignment (SSA) form [8] is an intermediate representation having the property that each variable—whether representing source-level variables chosen by the programmer or temporaries generated by the compiler—has exactly one definition point. Even though definitions in reentrant code may be visited many times in the course of execution, statically each SSA variable is assigned exactly once. If several distinct assignments to a variable occur in the source program, the building of SSA form splits that variable into distinct versions, one for each definition. In basic blocks where execution paths merge (such as the bottom block in our example) the consequent merging of variables is represented by a *phi function*. Phi functions occur only in instructions at the beginning of a basic block (before all non-phi instructions) and have the same number of operands as the block has predecessors in the CFG, each operand corresponding to one of the predecessors. The result of a phi function is the value of the operand associated with the predecessor from which control has come. A phi function is an abstraction for moves among temporaries that would occur at the end of each predecessor. SSA form does not allow such explicit moves since all would define the same variable. For simplicity we refer to phi functions and the instructions that contain them as ϕs. Figure 1(c) shows the SSA form of the original program considered above.

SSA is desirable for many transformations and analyses because it implicitly provides *def-use* information [8]. A *def* is a definition point of a variable. We say that the variable being defined is the *target* of that def. A *use* is any other occurrence of the variable in the program. For any given use of a variable, the definition relevant to that use is known immediately because a variable (by which we mean an SSA version of a variable) has a unique def; a given definition's set of uses is simply all places where the target of the definition is used. SSA also relieves the optimizer from depending on the programmer's choice of variable names and the way variables are reused in the program, since a transformation may regard each version of a variable as a completely different variable.

As a final preliminary, we explain the distinction between *availability* and *anticipation* (sometimes called *anticipability*) [1,5]. In general, we say that an expression is available at a program point if it has been computed on all paths to that point, and that an expression is anticipated at a point if it is computed on all paths from that point to program exit. There may be more restrictions based on such things as a programming language's memory model, but these definitions will suit our purposes. In Figure 1(c), $a_3 + b$ is anticipated at the beginning of block 3. Since a_3 is the same as a_1 on the edge (1,3), $a_1 + b$ is anticipated at the end of block 1. Because of the computation of $a_1 + b$ in block 1, that expression is also available at the end of the block. Since no equivalent expression is available at the end of block 2, $a_3 + b$ is not available at the beginning of block 3. Since the equivalent $a_1 + b$ is available from block 1, we say that $a_3 + b$ is *partially available* at the beginning of block 3. Similarly we can speak of *partial anticipation* if an expression is anticipated at the beginning of some but not all successors of a block.

SSAPRE

In this section, we explain the SSAPRE algorithm of Chow, Kennedy and co-workers [2,3], giving part of the motivation for the present work. First, we summarize the concepts and describe the algorithm's phases with an example. Second, we demonstrate weaknesses of the algorithm to show where improvements are necessary.

Summary

One useful tool when arranging any set of optimizations is a suitable, common program representation. If all optimizations use and preserve the same IR properties, then the task of compiler construction is simplified, optimization phases are easily reordered or repeated, and expensive IR rebuilding is avoided. Despite SSA's usefulness as an IR, prior work on PRE not only had not used SSA, but in the case of Briggs and Cooper [9], explicitly broke SSA before performing PRE. The principle motivation for SSAPRE is to do PRE while taking advantage of and maintaining SSA form.

SSAPRE associates expressions that are lexically equivalent when SSA versions are ignored. For example, $a_1 + b_3$ is lexically equivalent to $a_2 + b_7$. We take $a + b$ as the canonical expression

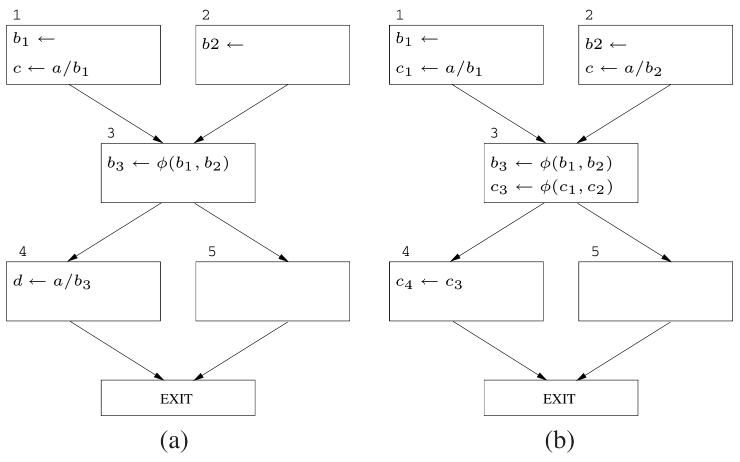

Figure 2. Unsafe code motion (a) unoptimized, (b) incorrectly 'optimized'.

and expressions like $a_1 + b_3$ as versions of that canonical expression; lexically equivalent expressions are assigned version numbers analogous to the version numbers of SSA variables. A *chi statement*[‡] (or simply χ) merges versions of *expressions* at CFG merge points just as ϕs merge variables. The χs can be thought of as potential ϕs for a hypothetical temporary that may be used to save the result of the computation if an opportunity for performing PRE is found. Just as a ϕ stores its result in a new version of the variable, so χs are considered to represent a new version of an expression. The operands of a χ (which correspond to incoming edges just as ϕ operands do) are given the version of the expression that is available most recently on the path they represent. If the expression is never computed along that path, then there is no version available, and the χ operand is denoted by $\perp$. χs and χ operands are also considered to be occurrences of the expression; expressions that appear in the code are differentiated from them by the term *real occurrences*.

For a code-moving transformation to be correct, it must never add a computation to a control path. (Since PRE does not alter the structure of the CFG, control flow paths in the optimized program correspond to paths in the unoptimized program.) In other words, it is incorrect to insert a computation if there exists a path from the insertion point to program exit along which the computation is not used. Not only would this lengthen an execution path (defying optimality), but it might also cause the optimized code to throw an exception that the unoptimized code would not. Consider the programs in Figure 2. When unoptimized, the computation of a/b_3 in block 4 is partially redundant. If we 'optimize' it by inserting a computation of a/b_2 in block 2 and eliminating a/b_3 in block 4, we shorten the control flow path (1,3,4,Exit), but lengthen the path (2,3,5,Exit). What is more, if b_2 is zero, the

[‡]In the original description of SSAPRE [2,3], χs are called Phis (distinguished from phis by the capitalization) and denoted in code by the capital Greek letter Φ.

path (2,3,5,Exit) will now raise an exception. The property that a computed expression will not be introduced to a control flow path is called *downsafety*. A χ is downsafe if its value is used on all paths to procedure exit. This and other properties of χs and χ operands are summarized in Table I. Details about their use and how to compute them are discussed later.

SSAPRE has six phases. We will discuss these while observing the running example in Figure 3. The unoptimized program is in Figure 3(a). The expression $b + c$ in block 4 is partially redundant because it is available along the right incoming edge.

1. **χ Insertion** For any real occurrence of a canonical expression on which we wish to perform SSAPRE, insert χs at blocks on the dominance frontier of the block containing the real occurrence and at blocks dominated by that block that have a ϕ for a variable in the canonical expression. (These insertions are made if a χ for that canonical expression is not there already.) These χs represent places that a version of an expression reaches but where it may not be valid on all incoming edges and hence should be merged with the values from the other edges. In our example, a χ is inserted at the beginning of block 4 for canonical expression $b + c$.
2. **Rename** Assign version numbers to all real expressions, χs, and χ operands. This algorithm is similar to that for renaming SSA variables given in Cytron *et al.* [8]. While traversing the dominator tree of the CFG in preorder, maintain a renaming stack for each canonical expression. The item at the top of the stack is the defining occurrence of the current version of the expression. For each block in the traversal, inspect its χs, its real instructions, and its corresponding χ operands in the χs of its successors, assigning a version number to each. If an occurrence is given a new version number, push it on the stack as the defining occurrence of the new version. When the processing of a block is finished, pop the defining occurrences that were added while processing that block. Table II explains how to assign version numbers for various types of occurrences depending on the defining occurrence of the current version (this table is expanded from Table 1 of Kennedy *et al.* [3]). Note that χ operands cannot be defining occurrences. Figure 3(b) shows our example after Rename. The occurrence in block 3 is given the version 1. Since block 4 is on its dominance frontier, a χ for this expression is placed there to merge the versions reaching that point, and the right χ operand is given version 1. Since the expression is unavailable from the left path, the corresponding χ operand is $\perp$. The χ itself is assigned version 2. Since that χ will be on top of the renaming stack when $b + c$ in block 4 is inspected and since the definitions of its variables dominate the χ, it is also assigned version 2.
3. **Downsafety** Compute downsafety with another dominator tree preorder traversal, maintaining a list of χs that have not been used on the current path. When program exit is reached, mark the χs that are still unused (or used only by operands to other χs that are not *downsafe*) as not *downsafe*. In our example, the χ is clearly *downsafe*.
4. **WillBeAvail** Compute *canBeAvail* for each χ by considering whether it is *downsafe* and, if it is not, whether its operands have versions that will be available at that point. Compute *later* by setting it to false for all χs with an operand that has a real use for its version or that is defined by a χ that is not *later*. Compute *willBeAvail* by setting it true for all χs for which *canBeAvail* is true and *later* is false. Compute *insert* for each χ operand by setting it to true for any operand in a χ for which *willBeAvail* is true and either is $\perp$ or is defined by a χ for which *willBeAvail* is false. Since the χ in Figure 3(b) is *downsafe*, it can be available. Since its right operand has a

Table I. χ and χ operand properties.

Properties for χs	
downsafe	The value of the χ is used on all paths to program exit.
canBeAvail	None of the χ's operands are $\perp$ or the χ is downsafe (i.e. the value has been computed on all paths leading to this point or insertions for it can be made safely).
later	Insertions for the χ can be postponed because they will not be used until a later program point.
willBeAvail	The value of the χ will be available at this point after the transformation has been made; it can be available and cannot be postponed.
Property for χ operands	
insert	An insertion needs to be made so that this χ operand's value will be available.

version defined by a real occurrence, it cannot be postponed. Therefore, *willBeAvail* is true for the χ.

5. **Finalize** Using a table to record what use, if any, is available for versions of canonical expressions, insert computations for χ operands for which *insert* is true (allocating new temporary variables to store their value), insert ϕs in place of χs for which *willBeAvail* is true, and mark real expressions for reloading that have their value available at that point in a temporary. In Figure 3(c), we have inserted a computation for its $\perp$ χ operand and a ϕ in the place of the χ to preserve the value in a new temporary.
6. **CodeMotion** Replace all real expressions marked for reloading with a move from the temporary available for its version. A move from the new temporary added in the previous step can then replace the real occurrence in block 4, as Figure 3(c) displays.

Counterexamples

SSAPRE's concept of redundancy is based on source-level lexical equivalence. Thus it makes a stronger assumption about the code than SSA form: from what source-level variable an SSA variable comes. This dependence weakens SSAPRE's ability to find all redundancies. In the example above, since the definition of c is a move from a_1, $b + c$ has the same value as $b + a_1$ in block 2. This means inserting $b + c$ in block 2 is not optimal; although it is not redundant lexically, it clearly computes the same value. Situations like this motivate research for making (general) PRE more effective, such as in Briggs and Cooper [9]. In our example, all we need is a simple constant copy propagation to replace all occurrences of c with a_1, as shown in Figure 4(a). Now the expression in block 4 is fully redundant even in the lexical sense. SSAPRE is fragile when used with such a transformation because it assumes that no more than one *version* of a variable may be simultaneously live. Copy propagation breaks this condition here, since both a_1 and a_2 are live at the exit of block 3, and, assuming a_3 is used later in the program, both a_1 and a_3 are live in block 4. In this case, $b + a_1$ is given the same version in blocks 2

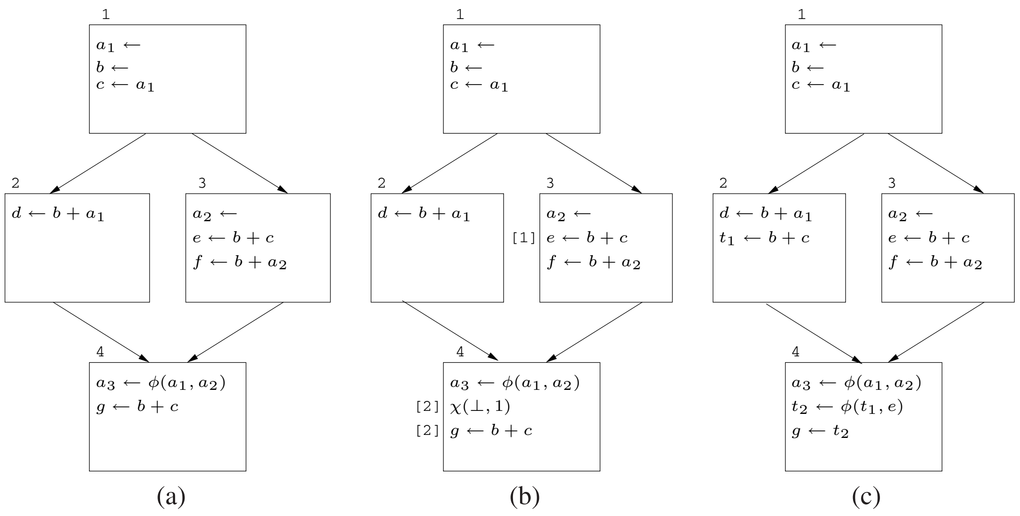

Figure 3. PRE example 1.

Table II. How to determine what version to assign to an occurrence of an expression.

	Occurrence being inspected		
Definition	real	χ	χ operand
real	Assign the old version if all corresponding variables have the same SSA version; otherwise assign a new version and push the item on the stack.	Assign a new version and push the item on the stack.	Assign the old version if the defining occurrence's variables have SSA versions current at the point of the χ operand; otherwise assign $\perp$.
χ	Assign the old version if all the definitions of the variables dominate the defining χ; otherwise assign a new version and push the item on the stack.	Assign a new version and push the item on the stack.	Assign the old version if the definitions of the current versions of all relevant variables dominate the defining χ; otherwise assign $\perp$.

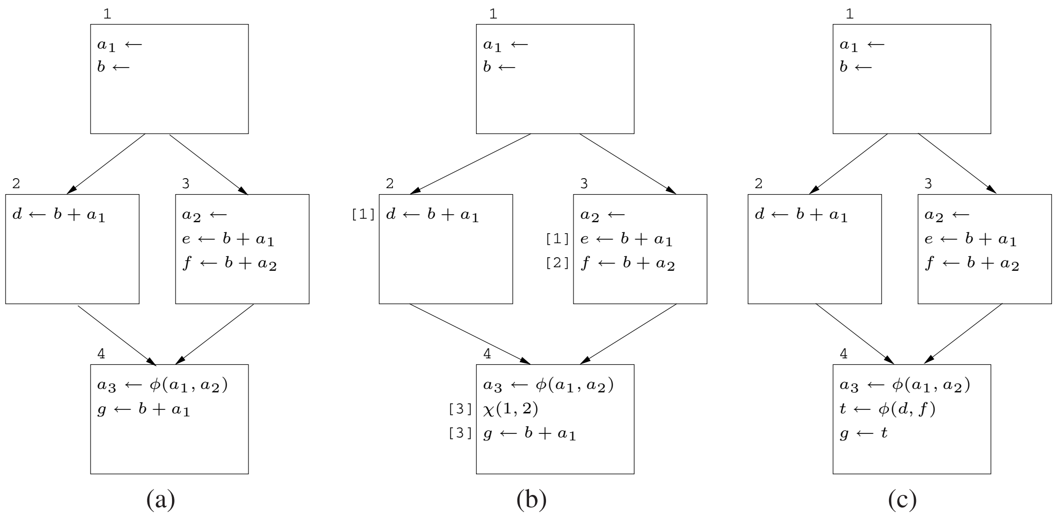

Figure 4. PRE example 1 after copy propagation.

and 3, while $b + a_2$ in block 3 is given its own version, since it does not match the current version $b + a_1$. See Figure 4(b).

What version should be given to the right operand of the χ in block 4? Version 2 ($b + a_2$) is the current defining occurrence, and Table II suggests that its version should be used if its operands are the current versions of the variables. Since a_2 is the corresponding operand to the ϕ at block 4, it can be considered current, and we give the χ operand version 2. However, this produces incorrect code. In the optimized program in Figure 4(c), g has value $b + a_2$ on the right path, not $b + a_1$ as it should. Chow, Kennedy and co-workers [2,3] give an alternative Rename algorithm called Delayed Renaming which might avoid producing wrong code. In that algorithm, the 'current' variable versions for a χ operand are deduced from a later real occurrence that has the same version as the χ to which the operand belongs, adjusted with respect to the ϕs at that block. In this case, the χ has the same version as $g \leftarrow b + a_1$. Since neither b nor a_1 are assigned by ϕs, they are the current versions for the χ operand, and are found to mismatch $b + a_2$; thus the χ operand should be $\perp$. However, this would still miss the redundancy between $e \leftarrow b + a_1$ and $g \leftarrow b + a_1$.

The entry in Table II for processing real occurrences when a χ is on the stack is also fragile. Figure 5 displays another program before, during, and after SSAPRE. Since two canonical expressions ($b + a$ and $b + c$) are being processed, we distinguish their χs and version labels with α and β, respectively. The same program, before SSAPRE but after constant copy propagation, is shown in Figure 6(a). There is now only one canonical expression. χ Insertion and Rename produce the version in Figure 6(b). The right χ operand is $\perp$ because a_1 in $b + a_1$ is not 'current' on that edge, not being the corresponding operand to the ϕ. Such a χ represents the correct value for $f \leftarrow b + a_3$. However, $g \leftarrow b + a_1$ is also assigned the same version: the χ is the definition of the current version, and the

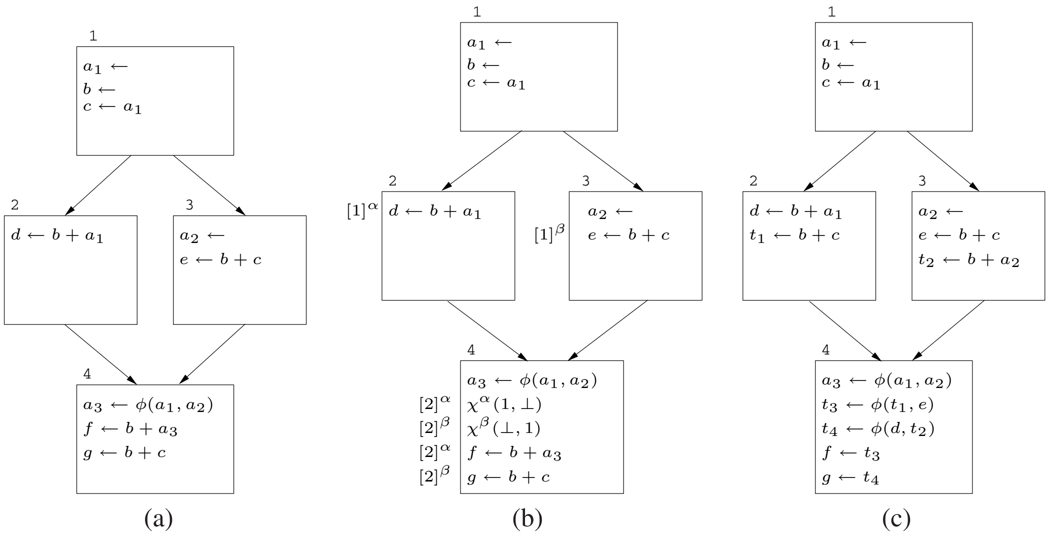

Figure 5. PRE example 2.

definitions of all the operands of $g \leftarrow b + a_1$ dominate it, which are the conditions prescribed by Table II. Consequently, the optimized version assigns an incorrect value to g in (c).

In both cases, the problem comes from the algorithm assuming that only one version of a variable can be live at a given time. Simple fixes are conceivable, but not without liabilities. Renaming variables to make SSAPRE's assumption valid will merely undo the optimizations intended to make PRE more effective. The rules in Table II could be made more strict; for example, we could associate versions of variables with each χ and make sure all versions match before assigning a real occurrence the same version as a χ. This would prevent the incorrect result in Figure 6, but since $b + a_1$ in the original is indeed partially redundant, ignoring it would not be optimal.

Another case when SSAPRE misses redundancy is when a redundant computation has subexpressions. Consider the program in Figure 7(a). The expression $a + b$ in block 4 is clearly partially redundant; $c_3 + d_2$ is also partially redundant, since d_1 and d_2 have the same value on that path. Essentially, it is the complex expression $c + a + b$ that is redundant.

Figure 7(b) shows the program after χ Insertion and Rename. As before, version numbers and χs for the two different expressions are distinguished by Greek letters. $d_2 \leftarrow a+b$ is given the same version as the corresponding χ and will be removed by later SSAPRE phases. However, $f \leftarrow c_3+d_2$ contains d_2, whose definition does not dominate the χ^β. According to Table II, we must assign the real occurrence a new version, even though the χ represents the correct value on the right path. The optimality claim of Chow, Kennedy and co-workers [2,3] assumes that a redundancy exists only if there is no non-ϕ assignment to a variable in the expression between the two occurrences; this example frustrates that assumption, and comparing this with the running example in Briggs and Cooper [9] demonstrates that

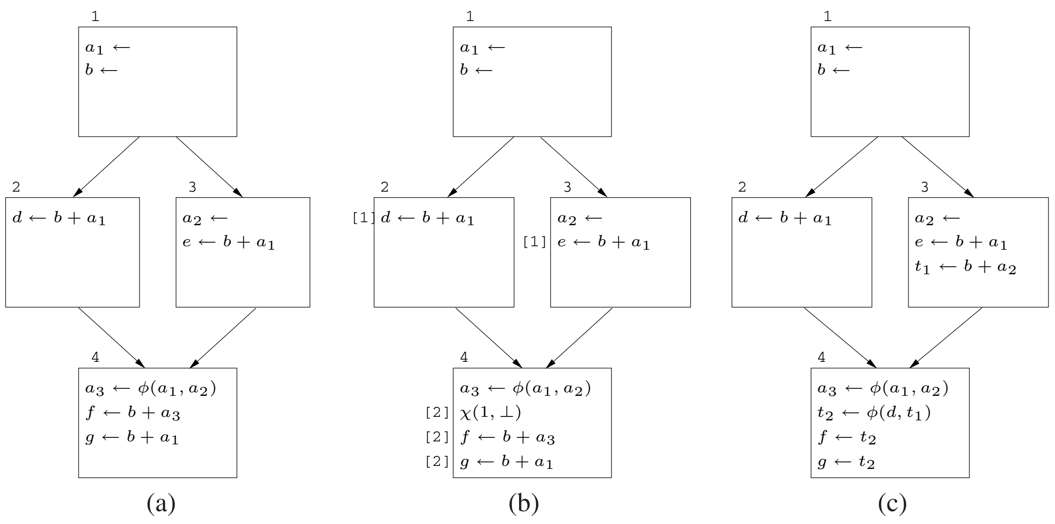

Figure 6. PRE example 2 after copy propagation.

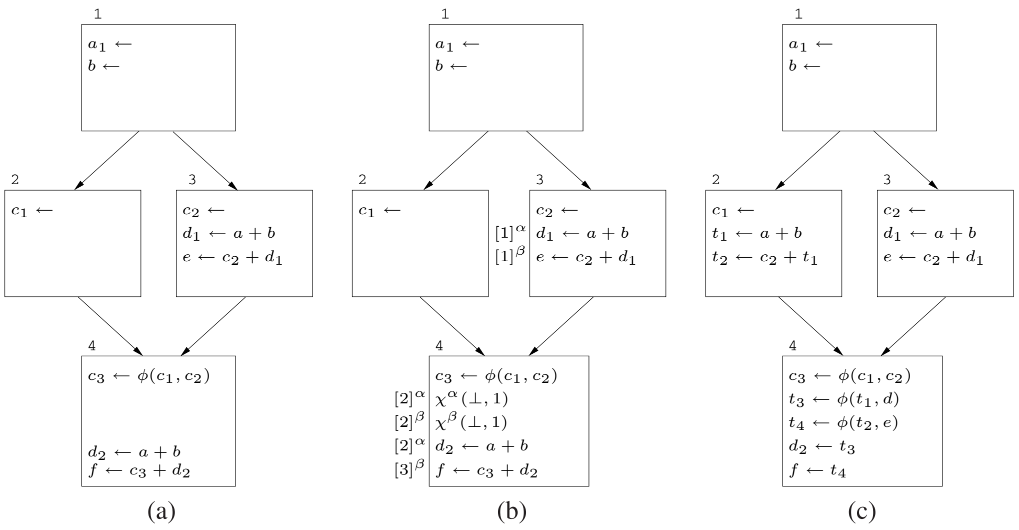
1
a1 ← b ←
2
c1 ←
3
c2 ← d1 ← a + b e ← c2 + d1
4
c3 ← φ(c1, c2) d2 ← a + b f ← c3 + d2
[1]α [1]β [2]α [2]β [3]β
χα(⊥, 1) χβ(⊥, 1)
t1 ← a + b t2 ← c2 + t1
t3 ← φ(t1, d) t4 ← φ(t2, e) d2 ← t3 f ← t4
(a)
(b)
(c)

Figure 7. Nested redundancy.

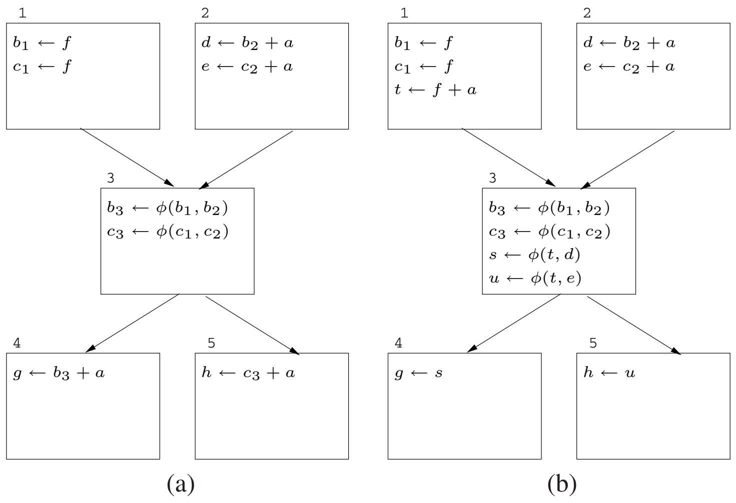

Figure 8. Redundancy eliminated by Bodík *et al.* [4,5].

this situation is not unrealistic when PRE is used with other optimizations. The optimized form we desire is in Figure 7(c), which would result if SSAPRE were run twice. (This situation is not just a product of our three-address representation, since it is conceivable that d has independent uses in other expressions.)

Finally, consider the example in Figure 8(a). The computations $b_3 + a$ and $c_4 + a$ in blocks 4 and 5, respectively, are partially redundant, since they are each computed in block 2. Thinking strictly lexically, we cannot insert computations in block 1 for either expression, because such a computation would not be downsafe, as it has a path to exit on which the expression is not computed. However, note that in block 1, b_2 and c_2 have the same value, and therefore $b_2 + a$ and $c_2 + a$ will also have the same value. Accordingly, only one computation need be inserted for both $b + a$ and $c + a$, and that computation would be downsafe because (if we think in terms of values and not just lexical expressions) it is used in both block 4 and block 5. The optimized version appears in Figure 8(b). Being blinded by lexical equivalence is only half the problem; we also need a way to recognize that the χs for expressions $b + a$ and $c + a$ have parameters from block 1 which need to be considered to be the same parameter for the sake of computing downsafety. Cases like this are discovered and eliminated by the path-based value-numbering of Bodík *et al.* [4,5].

A-SSAPRE

The problems with SSAPRE stem from assumptions about lexical equivalence. From this point on, we will abandon all notions of lexical equivalence among distinct SSA variables. We will ignore the fact that some SSA variables represent the same source-level variables and consider any variable to be an independent temporary whose scope is all instructions dominated by its definition. We will feel free to generate new temporaries as much as we need, since copy propagation and good register allocation should be able to clean up afterwards.

As already noted, SSAPRE is characterized by availability: χs are placed at the dominance frontier of a computation and thus indicate where a value is (at least partially) *available*. In this section, we present a new algorithm, *Anticipation*-SSAPRE (A-SSAPRE), which searches backward from computations and places χs at merge points where a value is *anticipated*. If the algorithm discovers the definition of a variable in the expression for which it is searching, it will alter the form of that expression appropriately; for example, if it is searching for $t_3 + t_7$ and discovers the instruction $t_7 \leftarrow t_1$, then from that point on it will look for t_3+t_1. (We do not need also to look for t_3+t_7 since t_7 is out of scope at any program point prior to its definition.) When a χ is created, it is allocated a fresh hypothetical temporary, which will be the target of the ϕ that will be placed there if the χ is used in a transformation. The result of this phase is that instructions are assigned temporaries that potentially store earlier computations of the instructions' values. The instructions may be replaced by moves from those temporaries.

Since all variables are distinct, so are all expressions. This obviates the need for the notion of canonical expressions and the need for version numbers. Thus our algorithm has no renaming phase. Not only do the problems in SSAPRE come from ambiguities in the specification of Rename, but also our implementation experience has found Rename to be particularly prone to bugs [10]. Next, the algorithm analyzes the χ operands that are $\perp$ to determine which ones represent *downsafe* insertions and analyzes the χs to determine which ones can be made available, which ones can be postponed, and, from these, which ones should be made into ϕs. This happens in the two phases Downsafety and WillBeAvail. Finally, where appropriate, insertions are made for χ operands, ϕs are put in the place of χs, and redundant computations are replaced with moves from temporaries. This phase, CodeMotion, subsumes the Finalize phase in SSAPRE. Thus, A-SSAPRE has four phases from SSAPRE's six.

The following subsections describe the four phases of A-SSAPRE in detail. We clarify each step by observing what it does on a running example. The unoptimized version is in Figure 9. Assignments with no right-hand side are assumed to come from sources we cannot use in A-SSAPRE. For example, t_1, t_2, and t_3 in block 1 may be parameters to the procedure, and t_9 in block 5 could be the result of a function call.

χ Insertion

χs, χ operands, and instructions

χ Insertion is the most complicated of the phases, and it differs greatly from the equivalent phase in SSAPRE. A χ is a potential ϕ or merge point for a hypothetical temporary, and it has an expression associated with it, for which the hypothetical temporary holds a pre-computed value. The hypothetical temporary is allocated at the time the χ is created, so it is in fact a real temporary from the temporary pool of the IR; it is hypothetical only in the sense that we do not know at the time it is allocated whether or not it will be used in the output program. The hypothetical temporary also provides a convenient way of referring to a χ, since the χ is the hypothetical definition point of that temporary. In the examples, a χ will be displayed with its operands in parentheses followed by its hypothetical temporary in parentheses. The expression it represents is written in the margin to the left of its basic block (see Figure 10).

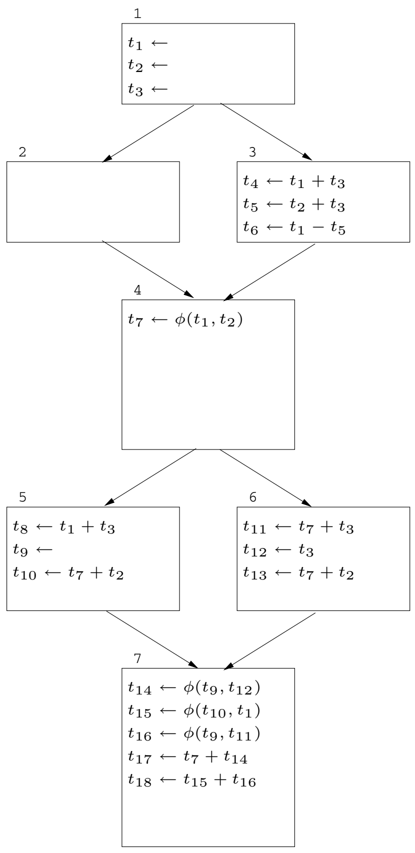
1
$t_1 \leftarrow$
$t_2 \leftarrow$
$t_3 \leftarrow$
2
3
$t_4 \leftarrow t_1 + t_3$
$t_5 \leftarrow t_2 + t_3$
$t_6 \leftarrow t_1 - t_5$
4
$t_7 \leftarrow \phi(t_1, t_2)$
5
$t_8 \leftarrow t_1 + t_3$
$t_9 \leftarrow$
$t_{10} \leftarrow t_7 + t_2$
6
$t_{11} \leftarrow t_7 + t_3$
$t_{12} \leftarrow t_3$
$t_{13} \leftarrow t_7 + t_2$
7
$t_{14} \leftarrow \phi(t_9, t_{12})$
$t_{15} \leftarrow \phi(t_{10}, t_1)$
$t_{16} \leftarrow \phi(t_9, t_{11})$
$t_{17} \leftarrow t_7 + t_{14}$
$t_{18} \leftarrow t_{15} + t_{16}$

Figure 9. Unoptimized.

A χ operand is associated with a CFG edge and either names a temporary or is $\perp$. They are represented in the examples accordingly, listed from left to right in the same order as their respective CFG edges appear. However, an important feature of our algorithm is that χs at the same block can share χ operands. Assume that if two χ operands are in the same block, are associated with the same CFG edge, and refer to the same temporary, then they are the same. χ operands that are $\perp$ are numbered to distinguish them. The temporary to which a χ operand refers is called the *definition* of that χ operand, and if the χ operand is $\perp$, we say that its definition is *null*. χ operands also have expressions associated with them, as explained below, but to reduce clutter, they are not shown in the examples.

Real occurrences of an expression (ones that appear in the code and are saved to a temporary) also have definitions associated with them, just as χ operands do. This is the nearest temporary (possibly hypothetical) found that contains the value computed by the instruction. If no such temporary exists,

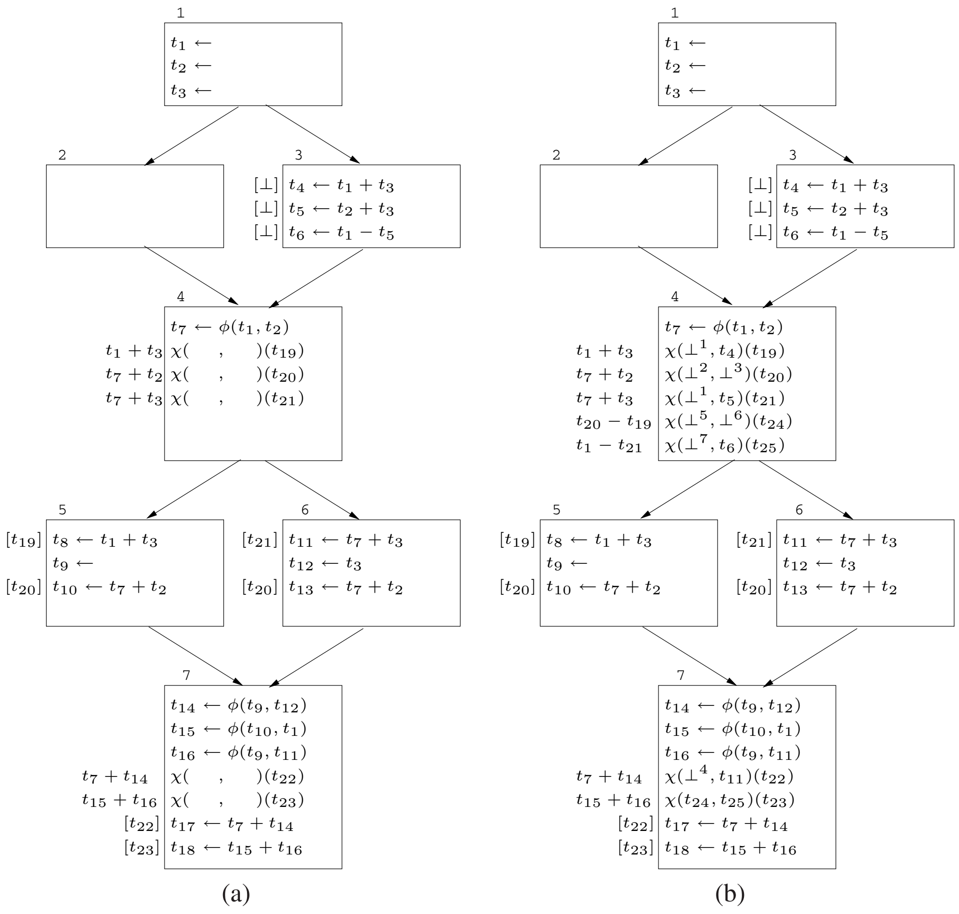

Figure 10. During χ Insertion.

then the definition is $\perp$. In the examples, this temporary is denoted in brackets in the margin to the left of its instruction. This notion of definition induces a relation among real instructions, χs, and χ operands and is equivalent to the *factored redundancy graph* of Chow, Kennedy and co-workers [2,3]

Searching from instructions

χ Insertion performs a depth-first, preorder traversal over the basic blocks of the program. In each basic block, it iterates forward over the instructions. For each instruction on which we wish to perform PRE (in our case, binary arithmetic operations), the algorithm begins a search for a definition. During the search, it maintains a search expression e, initially the expression computed by the instruction where it starts. It also maintains a reference to the original instruction, i. Until it reaches the preamble of the basic block (which we assume includes the ϕs and χs), the algorithm inspects an instruction at each step.

The current instruction being inspected is either a computation of e, the definition point of one of the operands of e, or an instruction orthogonal to e. In practice, we expect that this last case will cover the majority of the instructions inspected, and such instructions are simply ignored. If the instruction is a computation of e (that is, the expression computed exactly matches e), then the search is successful, and the temporary in which the current expression stores its result is given as the definition of i. When the current expression defines one of e's operands, then what happens depends on the type of instruction. If it is not itself a candidate for PRE or a simple move (for example, a function call), then nothing can be done; the search fails, and i's definition is null. If the current instruction is a move, then we emend e accordingly: the occurrences in e of the the target of the move are replaced by the move's source. The search then continues with the new form of e. If a copy propagation has been performed immediately before A-SSAPRE, then this should not be necessary, since moves will be eliminated. However, we make no assumptions about the order in which optimizations are applied, and in our experience such moves proliferate under many IR transformations.

If the current instruction stores the result of a PRE-candidate computation to an operand of e, then we can *conjecturally emend* the search expression; that is, we assume the instruction will be replaced by a move, and emend e as if that move were already in place. Since blocks are processed in depth-first preorder and instructions processed by forward iteration, we know that the instruction being inspected has already been given a definition. If that definition is not null, then we can conjecture that the instruction will be replaced by a move from the temporary of its definition to the temporary being written to, and we emend e as if that move were already present. From a software engineering standpoint, such conjectural emendations may be skipped during early development stages. However, they are necessary for optimality, since they take care of the situation in Figure 7. Without them, A-SSAPRE would require multiple runs to achieve optimality, especially in concert with the enabling optimizations described in Briggs and Cooper [9]. If the instruction's definition is null (or if conjectural emendations are turned off), such an instruction must be treated the same way as non-PRE candidates, and the search fails.

When the search reaches the preamble of a block, there are three cases, depending on how many predecessors the block has: zero (in the case of the entry block), one (for a non-merge point), or many (for a merge point). At the entry block, the search fails. We also do not hoist past an exception-handler header (we assume a *factored control flow graph* for representing exceptional control flow [20]). At a non-merge point, the search can continue starting at the last instruction of the predecessor block with no change to i or e. At a merge point, the algorithm inspects the expressions represented by the χs already placed in that block. If one is found to match e, then that χ (or more properly, that χ's hypothetical temporary) is given as i's definition. If no suitable χ is found, the algorithm creates one; a new temporary is allocated (which becomes i's definition), and the χ is appended to the preamble and to a list of χs whose operands need to be processed, as will be described below. In either case, the search is successful.

Consider the program in Figure 9. From the instruction $i \equiv t_4 \leftarrow t_1 + t_3$ in block 3, we search for an occurrence of $e \equiv t_1 + t_3$ and immediately hit the preamble of the block. Since it is not a merge point, the search continues in block 1. The write to t_3 is relevant to e, but since that instruction is not a move or a PRE candidate, the search fails and i is assigned $\perp$. Similarly, the searches from

$t_5 \leftarrow t_2 + t_3$ and $t_6 \leftarrow t_1 - t_5$ fail. For $i \equiv t_8 \leftarrow t_1 + t_3, e \equiv t_1 + t_3$, the search takes us to the preamble of block 4. Since it is a merge point, we place a χ there with expression $t_1 + t_3$ and allocate t_{19} as its hypothetical temporary, which also becomes the definition of $t_8 \leftarrow t_1 + t_3$. Similarly for $i \equiv t_{10} \leftarrow t_7 + t_3$, we place a χ, allocating the temporary t_{20}; and for $i \equiv t_{11} \leftarrow t_7 + t_3$, a χ with temporary t_{21}. For $i \equiv t_{13} \leftarrow t_7 + t_2, e \equiv t_7 + t_2$, when the preamble of block 4 is reached, we discover that a χ whose expression matches e is already present, and so its temporary (t_{20}) becomes i's definition. Finally, searches from the two real instructions in block 7 produce the χs with temporaries t_{22} and t_{23}, which serve as the real instructions' definitions. Figure 10(a) displays the program at this point.

Searching from χ operands

When all basic blocks in the program have been visited, the algorithm creates χ operands for the χs that have been made. Since χ operands have definitions just like real instructions, this involves a search routine identical to the one above. The only complications are χ-operand sharing among χs at the same block and emendations with respect to ϕs and other χs. For each χ c in the list generated by the search routine and for each in-coming edge η at that block, the algorithm determines an expression for the χ operand for that edge. To do this, we begin with the χ's own expression, e, and inspect the ϕs at the block and the χs that have already been processed. If any write to an operand of e (that is, if one of e's operands is the target of a ϕ or the hypothetical temporary of a χ), then that operand must be replaced by the operand of the ϕ that corresponds to η or the definition of the χ operand that corresponds to η. (If such a χ operand is $\perp$, then we can stop there, knowing that the χ operand we wish to create will be $\perp$, and that it is impossible to make an expression for it.) For example, if $e \equiv t_4 + t_6$ and the current block contains $t_6 \leftarrow \phi(t_1, t_2)$, the left χ operand will have the expression $t_4 + t_1$ and the right will have $t_4 + t_2$. We call the revised expression e'. Once an expression has been determined, the algorithm inspects the χ operands corresponding to η of all the χs at that block that have already been processed; if any such χ operand has an expression matching e', then that χ operand also becomes an operand of c. This sharing of χ operands is necessary to discover as many redundancies as Bodík [4], as the examples will show. If no such χ operand is found, the algorithm creates a new one with e' as its expression, and it searches for a definition for it in the same manner as it did for real instructions, in this case i referring to the χ operand. Note that this may also generate new χs, and the list of χs needing to be processed may lengthen.

So far, this phase has generated the five χs in Figure 10(a). The search for definitions for their operands is more interesting. In block 4, the t_{19} χ has expression $e \equiv t_1 + t_3$. Since none of e's operands are defined by ϕs, e also serves as the expression for the χ's operands. Search from the left operand fails when it hits the write to t_3 in block 1. Search from the right operand, however, discovers $t_4 \leftarrow t_1 + t_3$, which matches e; so t_4 becomes its definition. The t_{20} χ has expression $e \equiv t_7 + t_2$. Since $t_7 \leftarrow \phi(t_1, t_2)$ writes to one of e's operands, the two χ operands will have the expressions $t_1 + t_2$ and $t_2 + t_2$. Since neither have occurred in the program, two $\perp$ χ operands are created. The t_{21} χ has expression $e \equiv t_7 + t_3$. The ϕ changes this to $t_1 + t_3$ for the left edge and to $t_2 + t_3$ for the right edge. The former is identical to the expression for the left operand of the t_{19} χ, so that operand is reused. Searching from the right χ operand discovers $t_5 \leftarrow t_2 + t_3$.

Turning to block 7, the t_{22} χ has expression $t_7 + t_{14}$, which the ϕs change to $t_7 + t_9$ for the left edge and $t_7 + t_{12}$ for the right. On the left side, a search for a definition for the χ operand halts at the non-PRE candidate write to t_9 in block 5. The right hand search discovers $t_{11} \leftarrow t_7 + t_3$. The left and right χ operands for the t_{23} χ ($e \equiv t_{15} - t_{16}$) have expressions $t_{10} - t_8$ and $t_1 - t_{11}$, respectively. On the left, $t_{10} \leftarrow t_7 + t_2$ and $t_8 \leftarrow t_1 + t_3$ in block 5 affect the search expression, which we conjecturally emend to $t_{20} - t_{19}$. When the preamble of block 4 is reached, a new χ must be placed, allocating the temporary t_{24}. On the right, $t_{11} \leftarrow t_7 + t_3$ causes the search expression to be emended to $t_1 - t_{21}$, which also requires a new χ at block 4. Finally, we search for definitions for the χ operands of the two χs recently placed in block 4. The program at the end of the χ Insertion phase is shown in Figure 10(b).

Termination

Since the search for a definition for a χ operand may traverse a back edge, we now convince ourselves that the algorithm will terminate on loops; indeed, one qualification needs to be made for conjectural emendations in order to ensure termination. Consider the case when the search from a χ operand brings us back to the merge point of its χ. If the search expression has not changed, then the χ itself is the definition, and the search ends; otherwise, if no χ at that merge matches the current search expression, a new χ will be made. If operands in the search expression have been replaced only by other temporaries or constants in the input program (i.e. no conjectural emendations), then this process cannot go on forever, because there are only a finite number of temporaries or constants in the code. The danger lies when a temporary has been replaced by a hypothetical temporary. We say that a χ is *complete* if searches from all of its χ operands have been finished, and none refer to χs that are not complete. We say that a χ c *depends* on a χ operand ω if ω is an operand of c or if ω is an operand to a χ that in turn is referred to by a χ operand on which c depends. Supposing, as above, that e' is the search expression when searching for a definition of χ operand ω, then if the current instruction being inspected writes to an operand of e' but is defined by a χ c that does not depend on ω, then this search does not prolong the sequence of searches that make c complete. If c does depend on ω, we have no such guarantee. For example, if the expression in question is the increment of a loop counter, the algorithm would attempt to insert a χ for each iteration—an infinite number. To see this illustrated, consider Figure 11. t_1, t_2, and t_3 represent a loop-counter in the source code. If we search for an earlier occurrence of $t_2 + 1$, we will place a χ for it at the top of the block. Searching, then, for the χ operands, we emend the search expression to $t_3 + 1$ because of the ϕ. This leads us back to the original occurrence, $t_2 + 1$, with target t_3. If we were to conjecturally emend the search expression, we would replace t_3 with t_4, the occurrence's definition. Placing a χ, then for $t_4 + 1$, we search for an operand for the expression $t_5 + 1$, since the previous χ merges t_5 into t_4. Continuing like this would place χs at this point indefinitely. Thus, *when searching from a χ operand ω, conjectural emendations are forbidden if the current instruction is defined by a χ that depends on ω.*

Downsafety

Recall that an inserted computation is *downsafe* only if the same computation is performed on all paths to exit. Avoiding inserting computations that are not *downsafe* prevents lengthening computation paths and causing exceptions that would not be thrown in the unoptimized program. Since χ operands represent potential insertions, we consider downsafety to be a property of χ operands rather than of χs as in Chow, Kennedy and co-workers [2,3] and Table I. The second phase of A-SSAPRE is a static analysis that determines which χ operands are *downsafe*. We are interested only in χ operands that are

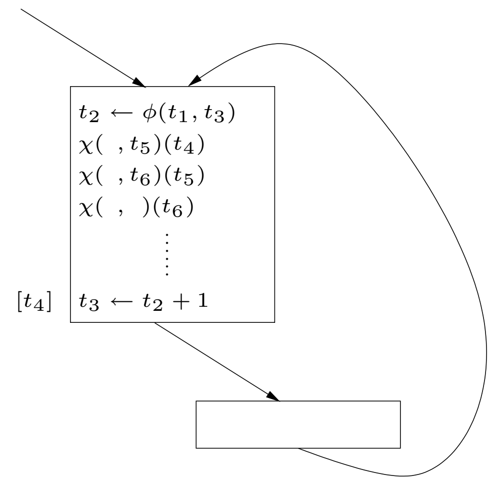

Figure 11. Unterminated χ Insertion.

$\perp$ or defined by a χ; if a χ operand's definition is the result of a real instruction, no insertion would ever need to be made for it. The question of whether a χ operand's value is used on all paths to program exit is akin to liveness analysis for variables and can be determined by a fixed-point iteration. At the entry to a block that is a merge-point, the χ operands of its χs become 'live'. A use of their value—which happens if a real instruction has the hypothetical temporary of the χ as a definition—'kills' them. All χ operands that are not killed in that block are 'live out', and must be considered 'live in' for the block's successors.

The fact that χs share χ operands complicates downsafety, since if a χ operand is an operand to several χs, a use of any of their hypothetical temporaries will kill it. Moreover, the use of a χ's hypothetical temporary will kill all of that χ's χ operands. To handle this, in addition to a set of live χ operands, we must maintain on entry and exit of every block a mapping from temporaries to sets of χ operands which a use of the temporaries will kill.

What if a χ operand's value is used as the definition to another χ operand? We could consider that χ operand to be a use occurring at the end of the corresponding predecessor block (not in the block that contains its χ). That would complicate things because if the χ operand was the only killer for the first χ operand on a path to exit, then the downsafety of the first would depend on the downsafety of the second. SSAPRE handled this by propagating a false value for downsafety from χs already found to be false to χs that depended on them. We can handle this with our map structure: if a use of t_1 will kill χ operand ω_1, χ operand ω_2 has t_1 as its definition, and a use of t_2 will kill ω_2, then we say that t_2 will also kill ω_1. The effect this has is that the use of a temporary as the definition of a χ operand cannot itself kill another χ operand, but that the fate of both χ operands are linked on subsequent paths to exit. One thing should be noted, though: if a χ appears at the beginning of a loop and is the definition of another χ there, its operand will have the temporaries or both χs as killers. If the second χ turns

Table III. Downsafety for the running example.

$\perp^1$	*downsafe*: killed by $t_8 \leftarrow t_1 + t_3$ on the left path and $t_{11} \leftarrow t_7 + t_3$ on the right
t_4	Irrelevant: defined by real instruction
$\perp^2$	*downsafe*: killed by $t_{10} \leftarrow t_7 + t_2$ on the left path and $t_{13} \leftarrow t_7 + t_2$ on the right
$\perp^3$	*downsafe*: killed by $t_{10} \leftarrow t_7 + t_2$ on the left path and $t_{13} \leftarrow t_7 + t_2$ on the right
t_5	Irrelevant: defined by real instruction
$\perp^5$	Not *downsafe*
$\perp^6$	Not *downsafe*
$\perp^7$	Not *downsafe*
t_6	Irrelevant: defined by real instruction
$\perp^4$	*downsafe*: killed by $t_{17} \leftarrow t_7 + t_{14}$
t_{11}	Irrelevant: defined by real instruction
t_{24}	*downsafe*: killed by $t_{18} \leftarrow t_{15} + t_{16}$
t_{25}	*downsafe*: killed by $t_{18} \leftarrow t_{15} + t_{16}$

out not to be inserted but is the definition of a real occurrence in the block, a χ operand in the first χ may be killed and the χ erroneously thought downsafe. To correct this, when a χ operand is new at the beginning of a basic block, only the χs that have it as an operand should define a kill for it.

Suppose *live_in*(b), *live_out*(b), *map_in*(b), and *map_out*(b) are the live in and out sets and mappings in and out, respectively, for a block b, and that Def(i) finds the definition of instruction i. Then to compute downsafety, we must solve the following data flow equations:

$$\begin{aligned}
\mathit{live\_in}(b) &= \Bigg(\bigcup_{p\in \mathit{pred}(b)} \mathit{live\_out}(p)\Bigg) \bigcup \{\chi \text{ operands at } b\}\\
\mathit{map\_in}(b) &= \mathit{clear}_b\Bigg(\bigcup_{p\in \mathit{pred}(b)} \mathit{map\_out}(p)\Bigg)\\
&\qquad \bigcup \{t \mapsto \sigma \mid \chi \text{ at } b \text{ with temporary } t \text{ and set of } \chi \text{ operands } \sigma\}\\
\mathit{live\_out}(b) &= \mathit{live\_in}(b) - \{\omega \mid \mathrm{Def}(i) = t \text{ and } \mathit{map\_in}(t) = \omega \text{ for some instruction } i \in b\\
&\qquad \text{or for some corresponding } \chi \text{ operand } i \text{ in a successor}\}\\
\mathit{map\_out}(b) &= \mathit{map\_in}(b)\\
\mathit{clear}_b m &= \{t \mapsto \sigma \mid t \mapsto \sigma' \in m \text{ and } \sigma = \sigma' - \Omega \text{ where } \Omega \text{ is the set of } \chi \text{ operands at } b\}
\end{aligned}$$

Table III lists what χ operands are *downsafe* in our example and why. As in the figures, χ operands are identified by their definition. Of particular interest is $\perp^1$ because it is *downsafe* only because it is shared—it is killed on the left path because it belongs to the t_{19} χ and on the right because it belongs to the t_{21} χ. If these were considered separate χ operands, then an opportunity to eliminate a redundancy would be missed.

WillBeAvail

The WillBeAvail stage computes the remaining properties for χs and χ operands, namely, *canBeAvail*, *later*, and *willBeAvail* for χs and *insert* for χ operands. The most important of these is *willBeAvail* because it characterizes χs that will be turned into ϕs for the optimized version.

We first determine whether it is feasible and safe for a χ to be made into a ϕ. If all of a χ's operands either have a real use or are *downsafe*, then that χ is *canBeAvail*. A χ is also *canBeAvail* if all of its χ operands that are not *downsafe* are defined by χs which also are *canBeAvail*, since, even though an insertion for that χ operand would not be safe, no insertion is needed if the value will be available from the defining χ.

To this end, after initializing *canBeAvail* to true for all χs in the program, we iterate through all χs. If a χ c has a χ operand that is not *downsafe* and is $\perp$ (and if *canBeAvail* has not already been proven false for c), we set *canBeAvail* to false for c. Then we make an inner iteration over all χs; for any χ that has an operand defined by c, if it has a non-*downsafe* χ operand but is still marked *canBeAvail*, then its *canBeAvail* should be cleared in the same manner. In our example, all χs are *canBeAvail* except for the ones with temporaries t_{24} and t_{25}, since they each have at least one $\perp$ χ operand that is not *downsafe*.

Next, we compute *later*, which determines if a χ can be postponed. This will prevent us from making insertions that have no benefit and would only increase register pressure. *later* is assumed true for all χs that are *canBeAvail*. Then we iterate through all χs, and if a χ c is found for which *later* has not been proved false and which has an operand defined by a real occurrence, we reset *later* for it. To do this, we not only set *later* to false, but, similarly to how *canBeAvail* was propagated, we also iterate through all χs; if any is found that has an operand with c as a definition, that χ's *later* is reset recursively. The idea is that if the definitions of any of a χ's variables are available (either because they are real occurrences or because they are χs that cannot be postponed), the χ itself cannot be postponed. In our example the t_{20} χ is *later* because both of its operands are $\perp$. The t_{23} χ is also *later* because both of its operands come from χs for which *canBeAvail* is false. These computations can be postponed.

At this point, computing *willBeAvail* is straightforward. A χ will be available if it can be made available and there is no reason for it not to be—that is, if it is *canBeAvail* and not *later*. In our example, all χs are *willBeAvail* except for the ones associated with t_{20}, t_{24}, and t_{25}. From this we also can compute *insert*, which characterizes χ operands that require a computation to be inserted. A χ operand is *insert* if it belongs to a *willBeAvail* χ and is either $\perp$ or defined by a χ for which *willBeAvail* is false. Being in a *willBeAvail* χ implies that such a χ operand is *downsafe*. In our example, χ operands $\perp^1$ and $\perp^4$ are *insert*. *willBeAvail* and *insert* can be computed on demand when they are needed in the next phase.

CodeMotion

The earlier phases having gathered information, the final stage, CodeMotion, transforms the code by inserting ϕs and anticipated computations and eliminating redundant computations. The net effect is to hoist code to earlier program points.

If *willBeAvail* is true for a χ, then the value it represents should be available in a temporary in the optimized program; a ϕ needs to be put in its place to merge the values on the incoming paths. The operands to this new ϕ will be the temporaries that hold the values from the various predecessors. If *insert* is true for any of its operands (indicating that the value it represents is not available, that is, has not been computed and stored in a temporary), then a computation for that value must be inserted at the end of the predecessor block it represents. Any real occurrence whose definition is another real occurrence or a *willBeAvail* χ is redundant, and can be replaced with a move from the temporary holding its value—if it is defined by a χ, the temporary is the target of the ϕ put in place of the χ; if it is defined by a real occurrence, the temporary is the one for the result of that occurrence.

Four steps complete the changes to the code: inserting appropriate computations, creating new ϕs, and eliminating fully redundant computations.

To do the insertions, we iterate over all χ operands. If any is marked *insert*, then we allocate a new temporary, manufacture an instruction which computes the χ operand's expression and stores the result in the fresh temporary, append that instruction at the end of the corresponding basic block, and set the fresh temporary to be the χ operand's definition. In our example, $\perp^1$ requires us to insert $t_{26} \leftarrow t_1 + t_3$ in block 2, where t_{26} is fresh. Similarly, we insert $t_{28} \leftarrow t_7 + t_9$ at block 5 for $\perp^4$.

We then iterate over all χs. For a χ c that *willBeAvail*, we insert a ϕ at the end of the list of ϕs already present at the block. That ϕ merges the temporaries that define c's χ operands into c's hypothetical temporary. Because insertions have been made, all valid χ operands will have temporaries for definitions by this point. In our example, we create $t_{19} \leftarrow \phi(t_{26}, t_4)$ and $t_{21} \leftarrow \phi(t_{26}, t_5)$ in block 4 and $t_{22} \leftarrow \phi(t_{28}, t_{11})$ in block 7.

Finally, we iterate over all instructions. If any is defined by the target of another real instruction or of a *willBeAvail* χ (which by this time has been made into a ϕ), it is replaced with a move instruction from its definition to its target. In our example, $t_{11} \leftarrow t_7 + t_3$ in block 6 is replaced with $t_{11} \leftarrow t_{21}$, and $t_{17} \leftarrow t_7 + t_{14}$ in block 7 is replaced with $t_{17} \leftarrow t_{22}$.

Final program

Figure 12 displays the final program. The improvement can be seen by comparing the number of computations on each possible execution path: (1,2,4,5,7), (1,3,4,5,7), (1,2,4,6,7), and (1,3,4,6,7). In the unoptimized program, the number of computations are four, four, seven, seven, respectively; in the optimized program, they are four, three, six, five. The benefit varies among the possible paths, but never is a path made worse.

Chow, Kennedy and co-workers claim that the running time of their algorithm is linear in the size of the program multiplied by the sum of the edges and nodes in the CFG [2,3]. One caveat with A-SSAPRE is that under certain conditions, the number of χs can explode—if a basic block has a large number of in-edges and a variable in one of that block's expressions has a different definition on each in-edge, the number of variant search expressions (and consequently χs) could become very large. While this is rare, we have seen some real code examples of methods with large and complicated CFGs where this algorithm is slow and space-hungry. We are addressing this concern in a successor algorithm to A-SSAPRE.

Other implementation concerns

There are two remaining implementation concerns we would like to address. First, extra care needs to be taken when applying this optimization to a language that supports precise handling of exceptions, such as Java. Instructions that could throw an exception (like any division where the denominator could be zero) must not be reordered, which could happen by the insertions and eliminations done by

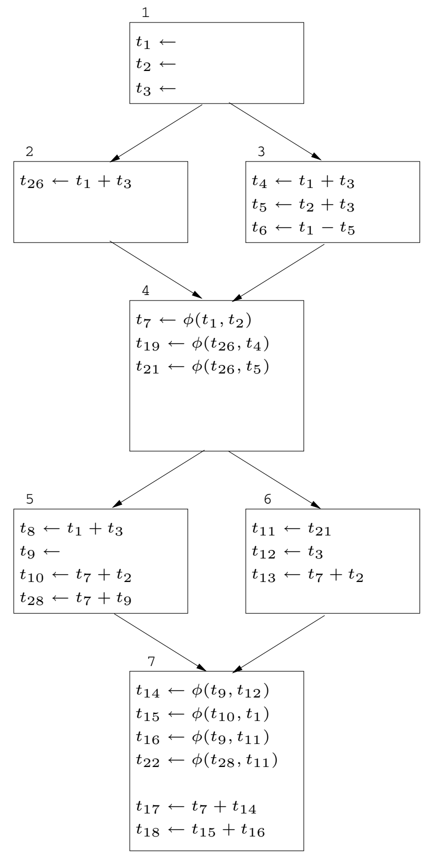
1
$t_1 \leftarrow$
$t_2 \leftarrow$
$t_3 \leftarrow$
2
$t_{26} \leftarrow t_1 + t_3$
3
$t_4 \leftarrow t_1 + t_3$
$t_5 \leftarrow t_2 + t_3$
$t_6 \leftarrow t_1 - t_5$
4
$t_7 \leftarrow \phi(t_1, t_2)$
$t_{19} \leftarrow \phi(t_{26}, t_4)$
$t_{21} \leftarrow \phi(t_{26}, t_5)$
5
$t_8 \leftarrow t_1 + t_3$
$t_9 \leftarrow$
$t_{10} \leftarrow t_7 + t_2$
$t_{28} \leftarrow t_7 + t_9$
6
$t_{11} \leftarrow t_{21}$
$t_{12} \leftarrow t_3$
$t_{13} \leftarrow t_7 + t_2$
7
$t_{14} \leftarrow \phi(t_9, t_{12})$
$t_{15} \leftarrow \phi(t_{10}, t_1)$
$t_{16} \leftarrow \phi(t_9, t_{11})$
$t_{22} \leftarrow \phi(t_{28}, t_{11})$
$t_{17} \leftarrow t_7 + t_{14}$
$t_{18} \leftarrow t_{15} + t_{16}$

Figure 12. Optimized.

this algorithm. We solve this elegantly by extending our downsafety algorithm. If we are checking the downsafety of a χ operand that represents an expression that could except, a use that occurs after another potentially excepting instruction should not make the χ operand considered downsafe. So when a potentially excepting instruction is encountered during downsafety, all potentially excepting χ operands should be marked not downsafe. The second concern is using this algorithm on object and array instructions. Including object references and array loads in the set of PRE instructions greatly increases the potency of A-SSAPRE, but care must be taken because objects and arrays can have aliases that make it difficult to know when two expressions are equivalent. In that case, it is best to rely on an underlying global value numbering, such as one that uses Array SSA form [11]. The search in χ-Insertion is for an expression (either a load or a store) that is definitely the same as the instruction from which we started. The search fails if we cross a store to something that is not definitely different,

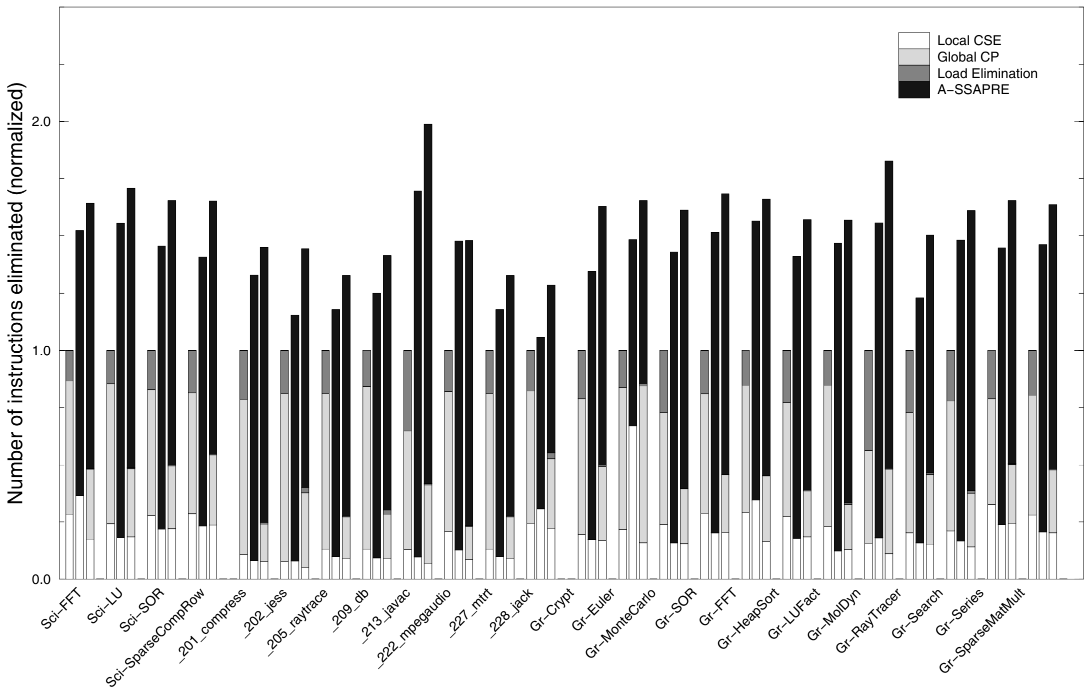

Figure 13. Static results for the benchmarks.

since that could represent a change in the expression's value. If reads kill [12], then this is true also for loads that are not definitely different.

EXPERIMENTS

Our experiments use JikesRVM [13,14], a virtual machine that executes Java classfiles. We have implemented the algorithm described here as a compiler phase for the optimizing compiler and configured JikesRVM version 2.2.0 to use the optimizing compiler only and a semi-space garbage collector.

The optimizing compiler performs a series of code transformations on both SSA and non-SSA representations. Before converting to SSA form, it performs branch optimizations, tail recursion elimination, dead code elimination, and constant folding. In SSA form, it performs local copy and constant propagation, Array SSA load elimination [11], global value numbering, and redundant branch elimination. After SSA form, it repeats earlier stages and performs common subexpression elimination and code reordering for better I-cache locality and branch prediction. We placed A-SSAPRE at the beginning of the SSA-based sequence. Our implementation performs the full optimization described in this paper and operates over not only arithmetic expressions but also null checks and array and object loads and stores.

We use three sets of benchmarks in our experiments: eight from the SPECjvm98 suite [15], four from SciMark [16], and 12 from the sequential benchmarks of the Java Grande Forum [17]. The runs were executed on a 733 MHz Power Macintosh with 32K I cache, 32 Kb D cache, and 256 Kb L2 cache running Mandrake Linux 8.2. Each benchmark run consists of 11 iterations, the first so that it will be fully compiled by the dynamic compiler and then (without restarting the virtual machine, so as to avoid recompiling the benchmark) run ten times, with the elapsed time measured for each run. The time is reported in milliseconds by Java's `System.currentTimeMillis()` call. Thus we avoid including compilation time in the measurement since compilation takes place in the initial, untimed run of each benchmark, since this experiment's goals were to show the effects of the transformation rather than to fine tune its implementation. We report the best of the final nine runs with garbage collection time subtracted out. We do this to isolate program performance and because we have found that heavy optimizations such as A-SSAPRE left the heap in such a state that later garbage collection times were greater during the run of the program than if the optimizations had not been performed.

Since A-SSAPRE is an expensive optimization, we study its effects on JikesRVM optimization level O2. We were particularly interested in how it competes against other optimizations that do code motion or eliminate computations. Accordingly, the four optimization phases we are concerned with are local common subexpression elimination (Local CSE), global code placement (Global CP, which includes global common subexpression elimination and loop invariant code motion), load elimination (Load Elimination), and the optimization presented here (A-SSAPRE). Given these optimization phases, we define the following three optimization levels.

O2: To mark our competition, we run the benchmarks at the pre-defined O2 level, which includes Local CSE, Global CP, and Load Elimination, but not A-SSAPRE.

PRE: To measure the effect of A-SSAPRE directly, we run A-SSAPRE without the other phases listed except for Local CSE (to clean up a few superfluous checks generated by our hoisting potentially excepting exceptions).

BOTH: To observe synergies, we run all four phases.

Figure 13 presents the static results of the optimization by giving the number of instructions eliminated by each optimization at each level. The bars for each benchmark from left to right represent the number of instructions eliminated at O2, PRE, and BOTH, respectively. The numbers are normalized so that the total number of instructions eliminated at O2 sum to 1. These results should be used only to give a feel for how much motion is being done, since this ignores the fact that at the same time instructions are also being inserted for hoisting. Therefore, each eliminated instruction, at least in the case of A-SSAPRE, should be considered as at least one execution path being shortened by one instruction. We feel that giving percentages of instructions eliminated would not provide more illumination, since many IR instructions do not map easily to machine instructions, and it would still hide the fact that some code is executed more frequently than other code. We see that A-SSAPRE can eliminate a larger number of instructions than Global CP and Load Elimination and subsumes much of what they do. In most cases, the number of instructions Load Elimination removes at the BOTH level is negligible. However, Load Elimination and Global CP were not completely subsumed. One reason for this, we found, was that in some cases (particularly for loop invariant code motion) an instruction

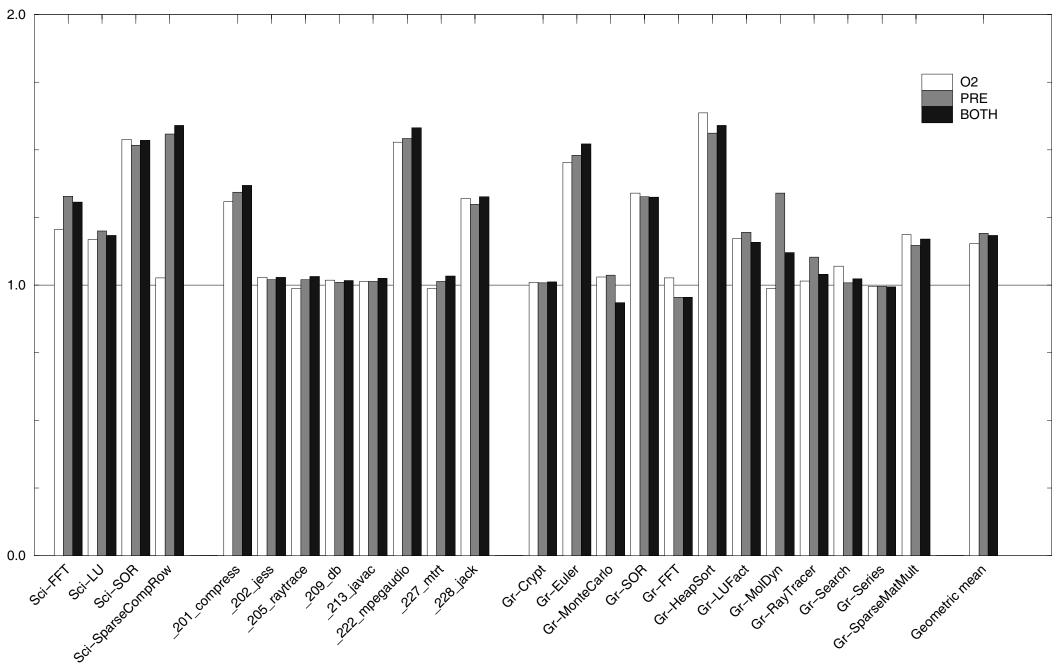

Figure 14. Speedup on benchmarks.

was being moved earlier or later in the loop without any redundant computation being eliminated or that loop invariant code motion was doing speculative motion that was not downsafe. The optimizations also had different constraints on what methods they could be applied to—since the optimizing compiler inlines very aggressively, some methods became too large to apply these optimizations practically, and there were some methods Global CP could run on that A-SSAPRE could not and *vice versa*. Inlining is a very important optimization because it eliminates many method calls and it has been found to be effective in improving run time [18].

We also considered impact on performance. Figure 14 shows speedups for the three optimization levels over a fourth level: O2 with all four optimizations turned off. Removing redundant instructions does not always significantly impact performance. However, on certain benchmarks we achieved speedups of up to 1.6. A-SSAPRE does about as well as global code placement and load elimination combined, and in many cases slices off a little more execution time. On two benchmarks (SciMark's SparseCompRow and Grande's MolDyn), it performs significantly better than its rivals.

In summary, we have demonstrated that A-SSAPRE is a complete PRE algorithm which competes with common subexpression elimination, loop invariant code motion, and load elimination, both statically and dynamically.

CONCLUSIONS AND FUTURE WORK

The contribution of this work is a simpler version of SSAPRE that makes fewer assumptions and covers more cases. The error-prone Rename phase has been eliminated, downsafety has been recast as a version of a standard data flow problem, the algorithm no longer makes assumptions about the namespace, and a wider range of redundancies are eliminated. It is now fit to be used in conjunction with other optimizations. By removing the dependence of SSAPRE on source-level lexical equivalence, we have moved it one step closer to Global Value Numbering (GVN). While PRE and GVN by themselves are not completely comparable (see section 12.4 of [19]), we are currently working on an algorithm that uses both availability and anticipation to make a hybrid of the two, covering the cases of both.

REFERENCES

1. Morel E, Renvoise C. Global optimization by supression of partial redundancies. *Communications of the ACM* 1979; **22**(2):96–103.
2. Chow F, Chan S, Kennedy R, Liu S-M, Lo R, Tu P. A new algorithm for partial redundancy elimination based on SSA form. *Proceedings of the Conference on Programming Language Design and Implementation*, Las Vegas, NV (*SIGPLAN Notices* 1997; **32**(5):273–286). ACM Press: New York, 1997.
3. Kennedy R, Chan S, Liu S-M, Lo R, Tu P, Chow F. Partial redundancy elimination. *ACM Transactions on Programming Languages and Systems* 1999; **21**(3):627–676.
4. Bodík R, Anik S. Path-sensitive value-flow analysis. *Proceedings of the Symposium on Principles of Programming Languages*, San Diego, CA, January 1988. ACM Press: New York, 1998; 237–251.
5. Bodík R, Gunta R, Soffa ML. Complete removal of redundant expressions. *Proceedings of the Conference on Programming Language Design and Implementation*, Montreal, Quebec (*SIGPLAN Notices* 1998; **33**(5):1–14). ACM Press: New York, 1998.
6. Aho AV, Sethi R, Ullman JD. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley: Reading, MA, 1986.
7. Knoop J, Rüthing O, Steffen B. Lazy code motion. *Proceedings of the Conference on Programming Language Design and Implementation*, San Francisco, CA (*SIGPLAN Notices* 1992; **27**(7):224–234). ACM Press: New York, 1992.
8. Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadek FK. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 1991; **13**(4):451–490.
9. Briggs P, Cooper KD. Effective partial redundancy elimination. *Proceedings of the Conference on Programming Language Design and Implementation*, Orlando, FL (*SIGPLAN Notices* 1994; **29**(6):159–170). ACM Press: New York, 1994.
10. Hosking AL, Nystrom N, Whitlock D, Cutts Q, Diwan A. Partial redundancy elimination for access path expressions. *Software Practice Experience* 2001; **31**(6):577–600.
11. Fink S, Knobe K, Sarkar V. Unified analysis of array and object references in strongly typed languages. *Proceedings of the Static Analysis Symposium*, Santa Barbara, CA, July 2000 (*Lecture Notes in Computer Science*, vol. 1824). Springer, 2000; 155–174.
12. Pugh W. Fixing the Java memory model. *ACM 1999 Java Grande Conference*. ACM Press: New York, 1999; 89–98.
13. Alpern B *et al.* The Jalapeño virtual machine. *IBM Systems Journal* 2000; **39**(1):211–238.
14. Burke MG, Choi J-D, Fink S, Grove D, Hind M, Sarkar V, Serrano M, Sreedhar V, Strinvasan H, Whaley J. The Jalapeño dynamic optimizing compiler for Java. *ACM 1999 Java Grande Conference*, June 1999. ACM Press: New York, 1999; 129–141.
15. Standard Performance Evalution Council. SPECjvm 98 benchmarks, http://www.spec.org/osg/jvm98/ [30 June 2004].
16. Pozo R, Miller B. Scimark 2.0. http://math.nist.gov/scimark2/ [30 June 2004].
17. EPCC. The Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/javagrande/index_1.html [30 June 2004].
18. Lee H, Diwan A, Moss JEB. Understanding the behavior of compiler optimizations. *Technical Report CU-CS-972-04*, Department of Computer Science, University of Colorado at Boulder, 2004.
19. Muchnick S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann: San Francisco, 1997.
20. Choi J-D, Grove D, Hind M, Sarkar V. Efficient and precise modeling of exceptions for the analysis of Java programs. *Workshop on Program Analysis for Software Tools and Engineering*. ACM Press: New York, 1999; 21–31.