Operator Strength Reduction

KEITH D. COOPER
Rice University
L. TAYLOR SIMPSON
BOPS, Incorporated
and
CHRISTOPHER A. VICK
Sun Microsystems, Incorporated

Operator strength reduction is a technique that improves compiler-generated code by reformulating certain costly computations in terms of less expensive ones. A common case arises in array addressing expressions used in loops. The compiler can replace the sequence of multiplies generated by a direct translation of the address expression with an equivalent sequence of additions. When combined with linear function test replacement, strength reduction can speed up the execution of loops containing array references. The improvement comes from two sources: a reduction in the number of operations needed to implement the loop and the use of less costly operations.

This paper presents a new algorithm for operator strength reduction, called OSR. OSR improves upon an earlier algorithm of Allen, Cocke, and Kennedy [Allen et al. 1981]. OSR operates on the static single assignment (SSA) form of a procedure [Cytron et al. 1991]. By taking advantage of the properties of SSA form, we have derived an algorithm that is simple to understand, quick to implement, and, in practice, fast to run. Its asymptotic complexity is, in the worst case, the same as the Allen, Cocke, and Kennedy algorithm (ACK). OSR achieves optimization results that are equivalent to those obtained with the ACK algorithm. OSR has been implemented in several research and production compilers.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*, *Optimization*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: loops, strength reduction, static single assignment form

1. INTRODUCTION

Operator strength reduction is a transformation that a compiler uses to replace costly (strong) instructions with cheaper (weaker) ones. A weak form of strength

This work was supported by DARPA, by IBM Corporation, and by Texas Instruments, Inc. DARPA provided much of the funding for the underlying project, the Massively Scalar Compiler Project.
Authors' addresses: K. D. Cooper, 6100 Main Street, MS 132, Houston, TX 77005; email: cooper@rice.edu; L. T. Simpson, 11129 Miramar Dr., Austin, TX 78726; email: taylor.simpson@austin.rr.com; C. A. Vick, Sun Micro Systems, Inc., 901 San Antonio Road, USCA 14-102, Palo Alto, CA, 94303-4900; email: christopher.vick@sun.com.

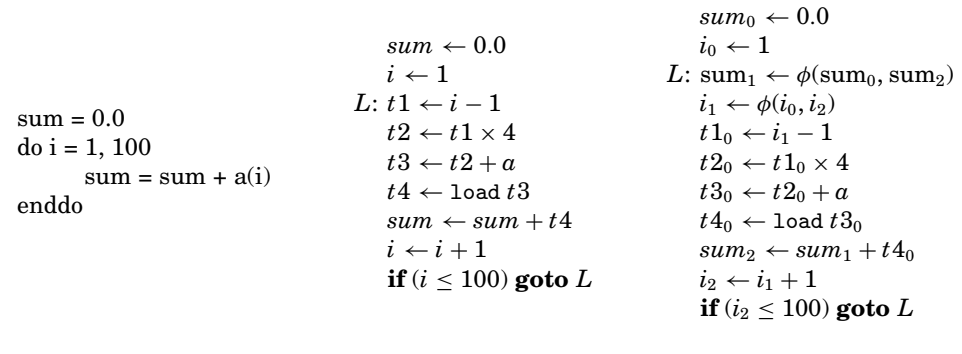

Fig. 1. Code for a simple loop.

reduction replaces $2 \times x$ with either $x + x$ or $x \ll 1$. The more powerful form of strength reduction replaces an iterated series of strong computations with an equivalent series of weaker computations. The classic example replaces certain repeated multiplications inside a loop with repeated additions. This case arises routinely in loop nests that manipulate arrays. The resulting additions are usually cheaper than the multiplications that they replace. In some cases, the additions can be folded into the target computer's addressing modes. Many operations other than multiplication can also be reduced in this manner. Allen, Cocke, and Kennedy provide a detailed catalog of such reductions [Allen et al. 1981].

This paper presents a new algorithm for performing strength reduction, called OSR, that improves upon the classic algorithm by Allen, Cocke, and Kennedy (ACK) [Allen et al. 1981]. By assuming some specific prior optimizations and operating on the SSA form of the procedure [Cytron et al. 1991], we have derived a method that is simple to understand and quick to implement. OSR achieves results that are, essentially, equivalent to those obtained with ACK, while avoiding some shortcomings of ACK, such as the need to apply ACK multiple times to reduce some of the induction variables created by other reductions. OSR's asymptotic complexity is, in the worst case, the same as the ACK algorithm.

Opportunities for strength reduction arise routinely from details that the compiler inserts to implement source-level abstractions. To see this, consider the simple Fortran code fragment shown in Figure 1. The left column shows source code; the middle column shows the same loop in a low-level intermediate code. Notice the four instruction sequence that begins at the label L. The compiler inserted this code (with its multiply) as the expansion of `a(i)`. The right column shows the code in *pruned SSA form* [Choi et al. 1991].

The left column of Figure 2 shows the code that results from applying OSR, followed by dead code elimination. The compiler created a new variable, $t5$, to hold the value of the expression $(i - 1) \times 4 + a$. Its value is computed directly, by incrementing it with the constant 4, rather than recomputing it on each iteration as a function of i. Strength reduction automates this transformation.

Of course, further improvement may be possible. For example, the only remaining use for i_2 is in the test that determines whether to terminate the loop

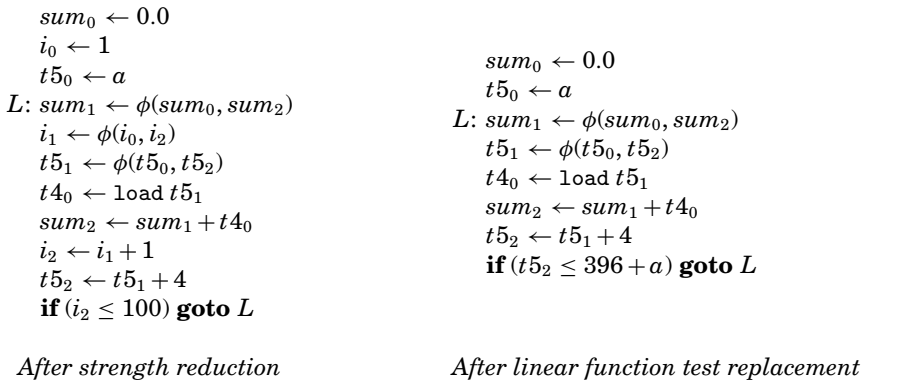

Fig. 2. Same loop after strength reduction.

or to continue for another iteration. The compiler can reformulate the tests to use $t5_2$, making the instructions that define i useless (or "dead"). This transformation is called *linear function test replacement* (LFTR). Applying this transformation, followed by dead code elimination, produces the code that appears in the right column of Figure 2.

Strength reduction has been an important transformation for two principal reasons. First, multiplying integers has usually taken longer than adding them. This made strength reduction profitable; the amount of improvement varied with the relative costs of addition and multiplication. Second, strength reduction decreased the "overhead" introduced by translation from a higher-level language down to assembly code. Opportunities for this transformation are frequently introduced by the compiler as part of address translation for array elements. In part, strength reduction's popularity stems from the fact that these computations are plentiful, stylized, and, in a very real sense, outside the programmer's concern.[1]

In the future, we may see microprocessors, where an integer multiply and an integer add, both take a single cycle. Even on such a machine, strength reduction will still have a role to play.

—Strength reduction often decreases the total number of operations in a loop. Smaller operation counts usually lead to faster code. The shorter sequences used to generate addresses may lead to tighter schedules, as well.

—In combination with algebraic reassociation [Cocke and Markstein 1980a; Santhanam 1992; Briggs and Cooper 1994], strength reduction can reduce the number of induction variables used in a loop, reducing the number of update operations required at the loop's end and reducing demand for registers.

[1]For dramatic evidence that the overhead computations introduced in translation are significant, look at the numbers given by Scarborough and Kolsky [1980] in their report on the Fortran H Extended compiler. Almost all of their improvements come from eliminating integer computations, not floating-point computations. Most of the eliminated instructions are introduced by translation to support the abstractions in the source-code program.

—On some machines, autoincrement or autodecrement features adjust a register's value as a side effect of a memory operation; strength reduction creates code that is shaped in a way to take advantage of this feature. With the multiply in place, the opportunity for autoincrement or autodecrement is hidden.
—Strength reduction decreases the number of multiplies and increases the number of additions. If more of the target machine's functional units can add than can multiply, this effect may give the scheduler additional freedom.

Thus, we expect that strength reduction will remain a useful transformation, even if the costs of addition and multiplication become identical.

The next section summarizes prior work on strength reduction, and places OSR in that context. We follow that section with a deeper introduction to strength reduction, given through a more detailed description of the ACK algorithm. Section 4 presents the OSR algorithm and its implementation. Section 5 shows how to fit linear function test replacement into our scheme.

2. PREVIOUS WORK

Reduction of operator strength has a long history in the literature. The first published discussions appear around 1969, in papers by Allen [1969] and Cocke and Schwartz [1970]. One family of algorithms grew out of the ideas in these seminal papers. A second family of algorithms grew out of the early work on data-flow-based optimization, typified by Morel and Renvoise's [1979] classic paper on partial redundancy elimination. A third body of work generalizes strength reduction to more complex domains than integer arithmetic. Finally, several authors have published papers that describe the implementations of the simpler, weaker form of strength reduction that applies knowledge about the values of operands to reduce an isolated instruction.

Allen-Cocke-Kennedy and its Descendants. A family of techniques has evolved from the early work described by Allen [1969] and Cocke and Schwartz [1970]. This includes work by Kennedy [1973], Cocke and Kennedy [1977], and Allen et al. [1981]. These algorithms transform a single loop at a time. They work outward through each loop nest, making passes to build use-definition chains, to find loops and insert prolog blocks, to find loop-invariant values (called *region constants*) and induction variables. Finally, they perform the actual reduction and instruction replacement in another pass. The ACK algorithm must be repeated to handle some second order effects. LFTR requires a separate pass over each loop. Cocke and Markstein [1980b] showed a transformation for reducing certain division and modulo operations. Chase [1988] extended the ACK algorithm to reduce more additions. Markstein et al. [1994] described a sophisticated algorithm that combines strength reduction with reassociation. Sites [1979] looked at the related issue of minimizing the number of loop induction variables.

Data-flow Methods. A second family of techniques has grown up around the literature of data-flow analysis [Dhamdhere 1979; Issac and Dhamdhere 1980; Dhamdhere 1989; Knoop et al. 1993]. These methods incorporate the strengths of data-flow based optimization, particularly the careful code-placement

techniques developed for partial redundancy elimination [Morel and Renvoise 1979]. The data-flow methods for strength reduction require none of the control-flow analysis needed by ACK and its descendants. This forces them to use a much simpler notion of region constant—they detect only simple literal constants. Thus, they miss some opportunities that the ACK-style methods discover, such as reducing $i \times j$ where i is the induction variable of the innermost loop containing the instruction and j is an induction variable of an outer loop. These algorithms must be repeated to handle second-order effects. Their placement techniques avoid lengthening execution paths; algorithms in the ACK family, including our own, cannot make the same claim.

Generalizations of Strength Reduction. A number of authors have looked at generalizations of strength reduction that work on set operations as well as classical integer arithmetic. Early [1974] first looked at this problem; he called the transformation "iterator inversion". Fong [1979] builds on that work to generalize the discovery of induction variables [Fong and Ullman 1976] and to reduce set formers in SETL. Paige generalized Early's work to create algorithms for "formal differentiation" [Paige and Schwartz 1977; Paige and Koenig 1982]. This, in turn, led to work on multiset discrimination as a way of avoiding the worst case behavior of hashing in actual implementations [Cai and Paige 1991]. Liu and Stoller [1998] have worked on generalizations of strength reduction that incrementalize certain loop-based array computations.

Weak Strength Reduction. This simpler form of the transformation, where an operation such as $2 \times x$ is replaced with either $x + x$ or $x \ll 1$, is widely used. Bernstein [1986] presents an algorithm used in the IBM PL.8 compiler that replaces integer multiply operations with a known constant argument by a sequence of add, subtract, and multiply operations. Briggs distributed a more complete implementation of these ideas via the Internet [Briggs and Harvey 1994], which appears to have been used by others [Wu 1995]. Granlund [1995] implemented a similar technique in the Gnu C compiler. He also looked at replacing division with multiplication [Granlund and Montgomery 1994].

Our Algorithm. OSR properly belongs in the ACK family of algorithms. It inherits the strengths of ACK, including the enlarged notions of region constant and induction variable. At the same time, it capitalizes on the properties of SSA to simplify both the explanation and the implementation. The result is a small, robust technique that is easy to implement and to maintain.

3. THE ALLEN-COCKE-KENNEDY ALGORITHM

Because OSR is intended as a replacement for the ACK algorithm, we begin by describing that algorithm. This allows us to make detailed comparisons when we present OSR. It also serves as a detailed introduction to the techniques and problems of strength reduction.

ACK focuses on loops, or *strongly connected regions* (SCRs), in a procedure's control-flow graph (CFG). For the present purpose, an SCR is a set of basic blocks with the property that a path exists between any two blocks in the SCR. The compiler discovers SCRs using an appropriate technique, such as Tarjan's

[1974] flow-graph reducibility algorithm or Havlak's [1997] extensions to it. ACK requires that each SCR have a unique *landing pad*—a prolog block that is always executed prior to entry into the SCR. The landing pad provides a convenient place to insert code that must execute before the loop. If landing pads do not exist, ACK inserts them. ACK processes the SCRs in a loop nest "inside out", starting from the most deeply nested SCR and working outward.

Conceptually, the first step in performing strength reduction is to identify operations that can be reduced. ACK searches an SCR to find instructions whose operands are either *region constants* or *induction variables* with respect to some SCR $\mathcal{S}$.

—A variable v is a region constant with respect to $\mathcal{S}$ if v's value does not change inside $\mathcal{S}$. We denote the set of region constants for $\mathcal{S}$ as $\mathcal{RC}(\mathcal{S})$.

—A variable i is an induction variable with respect to $\mathcal{S}$ if every definition of i inside $\mathcal{S}$ is one of: the sum of an induction variable and a region constant, an induction variable minus a region constant, or a COPY operation where the source is an induction variable. We denote the set of induction variables for $\mathcal{S}$ as $\mathcal{IV}(\mathcal{S})$.

These two sets, $\mathcal{RC}(\mathcal{S})$ and $\mathcal{IV}(\mathcal{S})$ are the key to determining whether an instruction is a candidate for strength reduction. ACK assumes that all the SCRs have been identified and their $\mathcal{RC}$ sets computed prior to its start. In practice, each of these requires a separate pass over the code.

We present ACK as if it operated on a single SCR. The code, shown in Figure 3, reflects this decision. The main routine, `ACK`, takes two arguments: a strongly connected region, *SCR,* and a set of region constants, *RC*. Its first step is to compute $\mathcal{IV}(SCR)$. For an initial approximation, it uses the set of all names defined by one of the appropriate instructions ($+$, $-$, or COPY). To refine this set, it iterates through the SCR a second time and removes any name that is defined by an operation other than $+$, $-$, or COPY, or that has operands that are not in $\mathcal{IV}(SCR)$ or $\mathcal{RC}(SCR)$.

The next step is to initialize a worklist of instructions that are candidates for reduction. The algorithm instantiates the worklist in a set called CANDS. To simplify our discussion, we will restrict our attention to candidate instructions in the following forms:

$$x \leftarrow i \times j \qquad x \leftarrow j \times i \qquad x \leftarrow i \pm j \qquad x \leftarrow j + i$$

where $i \in \mathcal{IV}(SCR)$ and $j \in \mathcal{RC}(SCR)$. Allen et al. [1981] describe a variety of other reducible candidate instructions. These are straightforward extensions to the algorithm.

Once the initial round of candidate instructions has been found, the algorithm repeatedly removes an instruction from the CANDS set and reduces it. For each candidate instruction, ACK creates a temporary variable to hold the value that it computes. It uses a hash table to avoid creating redundant temporary names [Kennedy 1973]. In Figure 3, the function `getName` implements the hash table. A call to `getName` consumes an expression (an opcode and two operands) and produces a temporary name. The first time that `getName` sees

```
ACK(SCR, RC)
    IV ← ∅
    for each instruction "x ← y op z" in SCR
        if op ∈ {+, −, COPY}
            IV ← IV ∪ {x}
    changed ← TRUE
    while changed do
        changed ← FALSE
        for each instruction "x ← y op z" in SCR with x ∈ IV
            if y ∉ IV ∪ RC or z ∉ IV ∪ RC
                IV ← IV − {x}
                changed ← TRUE

    CANDS ← ∅
    for each instruction i in SCR of the form x ← iv × rc,
                x ← rc × iv, x ← iv ± rc, or x ← rc + iv
        CANDS ← CANDS ∪ {i}

    while CANDS ≠ ∅
        Select and remove an instruction i from CANDS – "x ← iv op rc"
        result ← getName(op, iv, rc)
        if result = x
            Delete i
        else
            Replace i with "x ← result"

        for each definition point p of either iv or rc reaching i
            if there is no definition for result in the macro block for p
                if p is in the prolog
                    Insert "result ← iv op rc" at end of prolog
                    Perform constant folding if possible
                else if p is of the form "iv ← k"
                    Insert "result ← k op rc" into the macro block for p
                    Add this instruction to CANDS
                else if p is of the form "iv ← k + l" with k ∈ IV and l ∈ RC
                    if op = ×
                        result1 ← getName(op, k, rc)
                        result2 ← getName(op, l, rc)
                        Insert the following sequence into the macro block for p:
                            result1 ← k op rc
                            result2 ← l op rc
                            result ← result1 + result2
                        Add the first two instructions to CANDS
                    else
                        result1 ← getName(op, k, rc)
                        Insert the following sequence into the macro block for p:
                            result1 ← k op rc
                            result ← result1 + l
                        Add the first instruction to CANDS
                    Perform constant folding if possible
```

Fig. 3. The Allen-Cocke-Kennedy algorithm.

an expression, it generates a temporary name. Subsequent calls with the same arguments return the name already generated for the expression.

The candidate instruction is replaced with a COPY operation from the temporary name associated with the expression on the candidate's right-hand side. Next, the algorithm must insert instructions to compute the value of that temporary. These must be placed immediately before each instruction that defines either the induction variable or the region constant. To find these insertion points, the algorithm follows the *use-definition* chains for each of the operands [Kennedy 1978]. It inserts the instructions needed to initialize or update the reduced temporary's value, as appropriate. To determine the specific instructions that it inserts, the algorithm examines both the definition site and the current candidate instruction. Some of these new instructions may themselves be candidates for reduction. These are added to the worklist. Some of the instructions placed on the worklist in this way are subsequently deleted when they are removed from the worklist and processed.

To manage the insertion of instructions cleanly and to prevent the algorithm from inserting duplicate updates, ACK introduces the notion of a *macro block*. The collection of all instructions inserted at a point p is called the macro block for p. ACK only inserts an instruction at p if there is no definition of the same variable in p's macro block. This simple check ensures that the algorithm terminates. As long as the macro block stays small, this search should be fast.

To illustrate how ACK operates, we will apply it to the intermediate code in the middle column of Figure 1. The $\mathcal{RC}$ set contains $\{1, 4, a, 100\}$. ACK determines that $\mathcal{IV} = \{i, t1\}$. It initializes CANDS to {"$t1 \leftarrow i - 1$", "$t2 \leftarrow t1 \times 4$"}. Assume that the algorithm processes the instruction defining $t1$ first. The call to `getName` creates a new temporary name, $t5$, to hold the value of $i - 1$. Next, ACK replaces the candidate instruction with "$t1 \leftarrow t5$." It follows the use-definition chains to the assignments that define i. The first such definition is in the prolog, so ACK inserts "$t5 \leftarrow 0$" at the end of the prolog (simplified from "$t5 \leftarrow i - 1$"). The next definition increments the value of i at the bottom of the loop. After folding constants, ACK inserts two instructions:

$$t5 \leftarrow i - 1$$
$$t5 \leftarrow t5 + 1$$

It adds the first instruction to the worklist and processes it next. This causes ACK to delete the instruction and follow the use-definition chains for i. ACK finds that the macro block for each definition already contains a definition of $t5$, so it does not add any more instructions.

Next, ACK removes "$t2 \leftarrow t1 \times 4$" from the worklist. It invents a new temporary, $t6$, to hold the value of $t1 \times 4$. It replaces the candidate instruction with "$t2 \leftarrow t6$" and follows the use-definition chains to the assignment of $t1$. ACK inserts the instruction "$t6 \leftarrow t5 \times 4$," and adds it to the worklist.

Processing "$t6 \leftarrow t5 \times 4$," creates a new temporary, $t7$, to hold the value of $t5 \times 4$. ACK replaces the candidate instruction with "$t6 \leftarrow t7$" and follows the use-definition chains for $t5$. The first definition is in the prolog, so ACK inserts "$t7 \leftarrow 0$" at the end of the prolog. The next definition increments the value

of $t5$ at the bottom of the loop. After constant folding, two instructions are inserted here:

$$t7 \leftarrow t5 \times 4$$
$$t7 \leftarrow t7 + 4$$

The first instruction is added to the worklist and processed next. ACK deletes the instruction and follows the use-definition chains for $t5$. Since the macro block for each definition already contains a definition of $t7$, the algorithm terminates.

This example illustrates three shortcomings of ACK. First, it inserted two instructions that were subsequently removed. The instructions "$t5 \leftarrow i - 1$" and "$t7 \leftarrow t5 \times 4$" were removed when the algorithm discovered that the left-hand side was the temporary name associated with the expression on the right-hand side. Second, it had to search each macro block before inserting an instruction. If the macro blocks grow large, this search might become expensive. The third shortcoming is more subtle. ACK has not yet reduced the instruction "$t3 \leftarrow t2 + a$". The variable $t2$ is now an induction variable; it was not one when ACK began. To reduce this instruction, the compiler must re-apply the algorithm to the transformed program. In this example, the unrecognized candidate instruction is an addition. In more complicated examples, ACK can leave behind more expensive unreduced operations. To ensure that ACK reduces all possible candidates, the compiler must repeatedly apply the algorithm until it finds no further reductions.

4. THE OSR ALGORITHM

In deriving a new strength reduction algorithm, our goal was to clarify and to simplify the process [Vick 1994]. We wanted an algorithm that was easy to describe, easy to implement, reasonably efficient, and produced results comparable to ACK. The OSR algorithm achieves these goals. Where ACK operates on the procedure's CFG, OSR works, primarily, with the SSA graph.

OSR is driven by a simple depth-first search of the SSA-graph, using Tarjan's [1972] strongly-connected component finder. The SCC-finder lets OSR discover induction variables in topological order and process them as they are discovered. As the SCCs are discovered, they are processed by a set of mutually-recursive routines that test for induction variables and region constants, and then reduce the appropriate operations.

4.1 Preliminaries

Like ACK, our algorithm assumes that some prior optimization has been performed. Done correctly, this can simplify strength reduction by encoding certain facts in the code's shape. For example, after invariant code has been moved out of loops, region constants are easily identified by looking at where they are defined. We assume that the compiler performs the following transformations before OSR:

(1) *Constant propagation*: Sparse conditional constant propagation has been applied to identify and fold compile-time constants [Wegman and Zadeck

1991]. This discovers a large class of constant values and makes them textually obvious. It can increase the size of both the $\mathcal{RC}$ set and the $\mathcal{IV}$ set.

(2) *Code motion*: Lazy code motion [Knoop et al. 1992; Drechsler and Stadel 1993; Knoop et al. 1994], or one of its successors [Gupta et al. 1998; Bodik et al. 1998] has been applied to accomplish both loop invariant code motion and common subexpression elimination.[2] Applying global reassociation and global renaming prior to code motion can increase the $\mathcal{RC}$ set as well [Briggs and Cooper 1994].

Constant propagation and code motion improve the results of strength reduction by rewriting known values as literal constants and by moving invariant code out of loops. Both of these can expose additional opportunities for strength reduction.

After code motion and constant propagation, the algorithm builds the pruned SSA form of the program [Choi et al. 1991; Cytron et al. 1991; Briggs et al. 1998]. In SSA form, each name is defined exactly once and each use refers to exactly one definition. To reconcile these rules, the SSA construction inserts a special kind of definition, called a ϕ-function, at those points where control flow brings together multiple definitions for a single variable name. After ϕ-functions have been inserted, it systematically rewrites the name space, subscripting original variable names to ensure a unique mapping between names and definition points. Pruned SSA form includes only ϕ-functions that are *live*—that is, ϕ-functions whose values are used.

OSR operates on a graph that represents the SSA-form. In the program's *SSA graph*, each node represents either an operation or a ϕ-function, and edges flow from uses to definitions. The SSA graph can be built from the program in SSA form by adding use-definition chains, which can be implemented as a lookup table indexed by SSA names. The SSA graph provides a sparse representation of traditional use-definition chains. It also allows a more efficient method for identifying induction variables and region constants. Figure 4 shows the SSA graph for our example.

Even though it operates on the SSA graph, OSR needs to test specific properties defined on the procedure's CFG. These tests are couched in terms of two relationships on the CFG: dominance [Lengauer and Tarjan 1979] and reverse-postorder (RPO) numbering [Kam and Ullman 1976]. Dominance information is computed during the SSA construction. Our compiler computes a reverse-postorder numbering during CFG construction. OSR assumes that it can efficiently map any node in the SSA graph back into the CFG node that contains the corresponding operation.

4.2 Finding Induction Variables and Region Constants

OSR uses simple and effective tests to identify induction variables and region constants. These two tests form the first half of the algorithm.

[2]Lazy code motion will not move conditionally executed code out of loops. The later algorithms will. This can lead to a larger $\mathcal{RC}$ set.

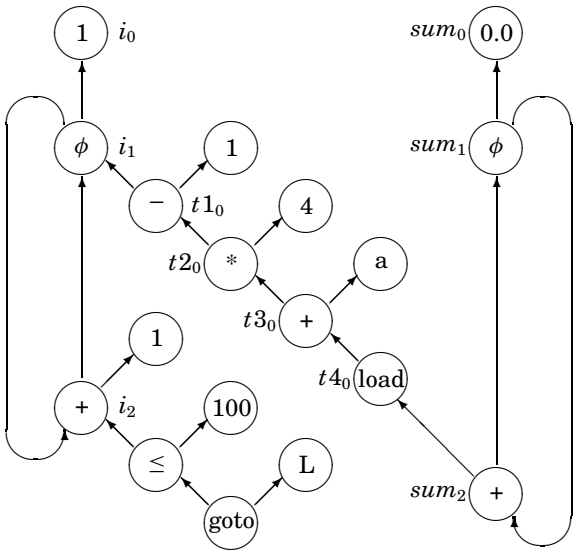

Fig. 4. SSA graph for the example.

4.2.1 *Induction Variables.* OSR finds induction variables by isolating and examining each strongly connected component (SCC) in the SSA graph. We differentiate between an SCC and an SCR, as used in the description of the ACK algorithm. In our description of ACK, we used SCR to mean a single loop in the CFG. An SCR may be nested inside other loops. An SCC for OSR is a maximal collection of nodes in the SSA graph with the property that a path exists between any two nodes in the SCC. These are related ideas, but with some critical distinctions.

—An SCR contains CFG nodes; thus, it may include updates to an arbitrary set of variables. Induction variables are found in SCRs, but the SCR also contains updates to other variables.

—An SCC contains SCA-graph nodes and edges; since these edges model the flow of values, the SCC describes a cyclic chain of dependences in the program.

Any induction variable, by definition, has a cyclic chain of SSA-graph edges. Thus, the set of SCCs in the SSA-graph must include all the induction variables.

OSR finds induction variables by inspecting each SCC. If all of the updates in the SCC have one of the allowed forms (*i.e.,* $\mathcal{IV} \pm \mathcal{RC}$, $\mathcal{IV} - \mathcal{RC}$, a COPY operation, or a ϕ-function[3]), then the SCC is an induction variable. Not every SCC represents an induction variable. In Figure 4, the SCC containing sum_1 and sum_2 does not represent an induction variable because $t4_0$ is not a region constant.

The code that finds induction variables is shown at the top of Figure 5. The routine, `ClassifyIV`, examines each operation in an SCC to see if it has one of the allowed forms. If the SCC is an induction variable, it labels each SSA node in the SCC with a *header* value. The header is the node in CFG that heads the

[3]For ϕ-nodes, each argument must be either a member of the SCC or a region constant.

```
ClassifyIV(SCC)
    for each n ∈ SCC
        if header→RPOnum > n.block→RPOnum
            header ← n.block
    for each n ∈ SCC
        if n.op ∉ {φ, +, −, COPY}
            SCC is not an induction variable
        else
            for each o ∈ {operands of n}
                if o ∉ SCC and not RegionConst(o, header)
                    SCC is not an induction variable
    if SCC is an induction variable
        for each n ∈ SCC
            n.header ← header
    else
        for each n ∈ SCC
            if n is of the form x ← iv × rc, x ← rc × iv, x ← iv ± rc, or x ← rc + iv
                Replace(n, iv, rc) (see Figure 7)
            else
                n.header ← NULL

RegionConst(name, header)
    return name.op = LOAD_IMMEDIATE or name.block ≫ header
```

Fig. 5. Finding region constants and induction variables.

outermost loop, or SCR, in which the SSA node is an induction variable. (This will be the CFG node associated with the SCC that has the smallest reverse-postorder number, *RPOnum*.) If the SCC is not an induction variable, it re-examines each node and either reduces it by calling `Replace` or labels it with the value `NULL` to indicate that it is not an induction variable. The header labels play a critical role in the test for region constants.

4.2.2 *Region Constants.* After constant propagation and code motion, OSR can use a simple test to determine if an operand is a region constant. An operand is a region constant if it is a compile-time constant, or if its definition strictly dominates every block in the CFG loop that contains the operation. (We use the notation $B_1 \gg B_2$ to denote that B_1 strictly dominates B_2.) Compile-time constants can be recognized syntactically, since constant propagation rewrites them in a standard form, such as a load immediate operation. For values that are defined outside the CFG loop and never modified inside it, the SSA construction ensures that the CFG block containing its definition will dominate the CFG block that is the loop's header.

These observations lead to a constant-time test for region constants, shown in the routine `RegionConst` at the bottom of Figure 5. It checks whether the operand results from a LOAD IMMEDIATE. It uses the labels computed during `ClassifyIV` to relate individual definitions in the SSA graph back to the CFG. If the CFG node containing the value's definition dominates the CFG node given by the header label of the value's use, then the value is loop invariant. (The dominance relation is computed during the SSA construction.) Because these

```
OSR(SSAgraph)
    while there is an unvisited node n in SSAgraph
        DFS(n) (see Figure 6)

ProcessSCC(SCC)
    if SCC has a single member n
        if n is of the form x ← iv × rc, x ← rc × iv, x ← iv ± rc, or x ← rc + iv
            Replace(n, iv, rc) (see Figure 7)
        else
            n.header ← NULL
    else
        ClassifyIV(SCC)

DFS(node)
    node.DFSnum ← nextDFSnum++
    node.visited ← TRUE
    node.low ← node.DFSnum
    PUSH(node)
    for each o ∈ {operands of node}
        if not o.visited
            DFS(o)
            node.low ← MIN(node.low, o.low)
        if o.DFSnum < node.DFSnum and o ∈ stack
            node.low ← MIN(o.DFSnum, node.low)
    if node.low = node.DFSnum
        SCC ← ∅
        do
            x ← POP()
            SCC ← SCC ∪ {x}
        while x ≠ node
        ProcessSCC(SCC)
```

Fig. 6. High-level code for OSR.

tests take constant time, it does not pay to instantiate the $\mathcal{RC}$ set. Thus, OSR tests for membership in $\mathcal{RC}$ on demand.

4.2.3 *Putting It Together.* To drive the entire process, OSR uses Tarjan's [1972] algorithm for finding SCCs. The algorithm, shown as routine DFS at the bottom of Figure 6, is based on depth-first search. It pushes nodes onto an internal stack as it visits them. The order in which the nodes are popped from the stack groups them into SCCs. A node that is not contained in any cycle is popped by itself, as a singleton SCC. The algorithm pops all the nodes in a multi-node cycle as a group, or a multi-node SCC.

Tarjan's algorithm has an interesting property: it pops the SCCs from the stack in topological order. Thus, when an SCC is popped from the stack, any operand referenced inside the SCC is either defined within the SCC itself or else it is defined in an SCC that has already been popped. OSR capitalizes on this observation and processes the nodes as they are popped from the stack. Thus, the SCC-finder drives the entire strength reduction process.

The top of Figure 6 shows the driver routine, OSR. It invokes DFS on each disjoint component of the SSA-graph. As DFS identifies each SCC, it invokes ProcessSCC on the SCC. If the SCC contains a single operation, ProcessSCC tries to reduce it—checking its form and invoking Replace (described in Section 4.3) if it can be reduced. Since this process transforms x into an induction variable, Replace labels x as an induction variable with the same header block as i. This allows further reduction of operations using x. If the SCC contains more than one operation, then ProcessSCC invokes ClassifyIV to determine if it is an induction variable and performs the appropriate reductions if it is not an induction variable.

The idea of finding induction variables as SCCs of the SSA graph is not original. Wolfe [1992] used it in his work on induction variables and dependence analysis, and suggested in the accompanying talk that the idea was obvious and had occurred to others in the field. Like OSR, Wolfe's work relies on the fact that Tarjan's algorithm discovers the SCCs in topological order.

4.3 Code Replacement

Once OSR has found a candidate instruction of the form $x \leftarrow i \times j$, it must update the code so that the multiply is no longer performed inside the loop. The compiler creates a new SCC in the SSA graph to compute the value of $i \times j$ and replaces the instruction with a COPY from the node representing the value of $i \times j$. This process is handled by three mutually recursive functions as shown in Figure 7:

- Replace rewrites the current operation with a COPY from its reduced counterpart.
- Reduce inserts code to strength reduce an induction variable and returns the SSA name of the result.
- Apply inserts an instruction to apply an opcode to two operands and returns the SSA name of the result. Simplifications such as constant folding are performed if possible.

The Replace function is straightforward. It provides the top-level call to the recursive function Reduce and replaces the current operation with a COPY. The resulting operation must be an induction variable.

The Reduce function is responsible for adding the appropriate operations to the procedure. The basic idea is to create a new induction variable with the same shape as the original, but possibly with a different initial value or a different increment. The first step is to check the hash table for the desired result. Access to the hash table is through two functions:

- search looks up an expression (an opcode and two operands), and returns the name of the result.
- add adds an entry containing an expression and the name of its result.

If the result is already in the hash table, then no additional instructions are needed, and Reduce returns the SSA name of the result. The single definition property of SSA lets OSR conclude that any name found in the table is already

```
Replace(node, iv, rc)
    result ← Reduce(node.op, iv, rc)
    Replace node with a COPY from result
    node.header ← iv.header

SSAname Reduce(opcode, iv, rc)
     result ← search(opcode, iv, rc)
    if result is not found
        result ← inventName()
        add(opcode, iv, rc, result)
        newDef ← copyDef(iv, result)
        newDef.header ← iv.header
        for each operand o of newDef
            if o.header = iv.header
                Replace o with Reduce(opcode, o, rc)
            else if opcode = × or newDef.op = φ
                Replace o with Apply(opcode, o, rc)
    return result

SSAname Apply(opcode, op1, op2)
    result ← search(opcode, op1, op2)
    if result is not found
        if op1.header ≠ NULL and RegionConst(op2, op1.header)
            result ← Reduce(opcode, op1, op2)
        else if op2.header ≠ NULL and RegionConst(op1, op2.header)
            result ← Reduce(opcode, op2, op1)
        else
            result ← inventName()
            add(opcode, op1, op2, result)
            Choose the location where the operation will be inserted
            Decide if constant folding is possible
            Create newOper at the desired location
            newOper.header ← NULL
    return result
```

Fig. 7. Code replacement functions.

defined. If the result is not found in the table, Reduce invents a new SSA name.[4] The copyDef function then copies the operation or ϕ-node that defines the induction variable and assigns the new name to the result.

Next, Reduce considers each argument of the new instruction. If the argument is defined inside the SCC, Reduce invokes itself recursively on that argument. (ACK handles this by inserting an operation and adding it to the worklist. This is the source of the operations that were added and then removed in the earlier example.)

Arguments defined outside the SCC are either the initial value of the induction variable or the value by which it is incremented. The initial value must be

[4]This hash lookup replaces the linear search through a macro block used by ACK to detect that it has already inserted the needed code.

an argument of a ϕ-node, and the increment value must be an operand of an instruction. The reduction is always applied to the initial value, but the reduction is only applied to the increment if we are reducing a multiply. In other words, when the candidate is an add or subtract instruction, our algorithm modifies only the initial value, but if the candidate is a multiply, it modifies both the initial value and the increment. Therefore, `Reduce` invokes `Apply` on arguments defined outside the SCC only if it is reducing a multiply, or if it is processing the arguments of a ϕ-node.

The `Apply` function is conceptually simple, although there are a few details that must be considered. The basic function of `Apply` is to create an operation that computes the desired result. `Apply` relies on the hash table to determine if such an operation already exists. It is possible that the operation `Apply` is about to create is a candidate for strength reduction in an enclosing loop. If so, it performs the reduction immediately by calling `Reduce`. (ACK inserts an instruction that gets reduced when the outer loop is processed.) This case often arises from triangular loops—where the bounds of an inner loop are a function of the index of an outer loop.

Before inserting the operation, the algorithm must select a legal location. ACK assumes that all induction variables are defined in the loop's prolog [Allen et al. 1981, p. 93]. Therefore, it inserts all initializations of temporaries at the end of the prolog block. Our algorithm relies on dominance information created during SSA construction to find a legal location for the initialization. Intuitively, the instruction must go into a block that is dominated by the definitions of both operands. If one of the operands is a constant, the algorithm may need to duplicate its definition to satisfy this condition. Otherwise, both operands must be region constants, so their definitions dominate the header block. One operand must be a descendant of the other in the dominator tree, so the operation can be inserted immediately after the descendant. This eliminates the need for landing pads; it may also place the operation in a less deeply nested location than the landing pad.

4.4 Applying OSR to the Example

As an example of how code replacement works, we will apply it to the SSA graph in Figure 4. `OSR` invokes `DFS` on the graph. No matter where it starts, `DFS` will find a small number of singleton SCCs, followed by the SCC $\{i_1, i_2\}$. It labels this SCC as an induction variable. Next, it finds the SCC containing just $t1_0$.

OSR identifies this operation as a candidate instruction, because one argument is an induction variable and the other is a region constant. It invokes `Replace` with $iv = i_1$ and $rc = 1$. The call to `search` in `Reduce` will fail, so the first SSA name invented will be osr_0. `Reduce` adds this entry to the hash table and creates a copy of the ϕ-node for i_1. Next, it processes the arguments of the new ϕ-node. Since the first argument, i_0, is a region constant, it is replaced with the result of `Apply`, which will perform constant folding and return the SSA name osr_1. The second argument, i_2, is an induction variable, so it invokes `Reduce` recursively.

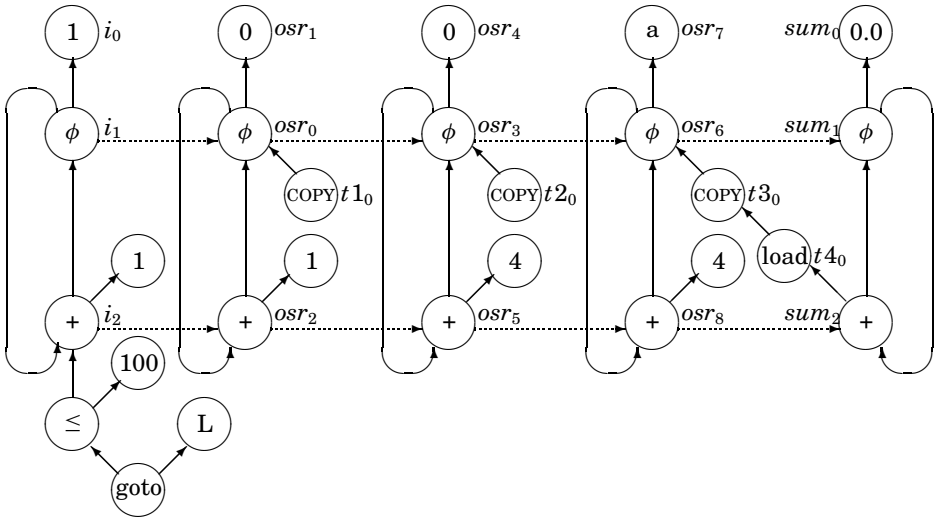

Fig. 8. SSA graph after applying OSR.

Since the hash table contains no match, `Reduce` invents a new SSA name, osr_2, adds an entry to the hash table, and copies the operation for i_2. The first argument is the region constant 1, which will be left unchanged. The second argument is i_1, which is an induction variable. The recursive call to `Reduce` will produce a match in the hash table with osr_0 as the result. At this point, the calls to `Reduce` finish, and the SSA name osr_0 is returned to `Replace`. `Replace` rewrites the operation defining $t1_0$ as a COPY from osr_0. It labels $t1_0$ as an induction variable to enable further reductions.

As subsequent SCCs pop from the stack, this same process creates two more reduced induction variables. The multiply labelled $t2_0$ gives rise to another new induction variable, as does the add labelled $t3_0$. The load operation cannot be reduced because it is not a candidate instruction. Similarly, the operations in the SCC that define *sum* are not candidates.

Figure 8 shows the SSA graph that results from applying OSR to our ongoing example program. The sequence of reductions unfolds from left to right, with the creation of the reduced induction variables involving osr_0, osr_3, and osr_6. The dashed gray lines show this derivation. The induction variables containing osr_0 and osr_3 are dead. The sole remaining use for the induction variable defining i is the $\leq$ operation that governs the branch.

4.5 Complexity of OSR

The time required to identify the induction variables and region constants in an SSA graph is $\mathbf{O}(N+E)$, where N is the number of nodes and E is the number of edges. The `Replace` function performs work that is proportional to the size of the SCC containing the induction variable, which can be as large as $\mathbf{O}(N)$. Since `Replace` can be invoked $\mathbf{O}(N)$ times, the worst case running time is $\mathbf{O}(N^2)$. This seems expensive; unfortunately, it is necessary. Figure 9 shows a program that

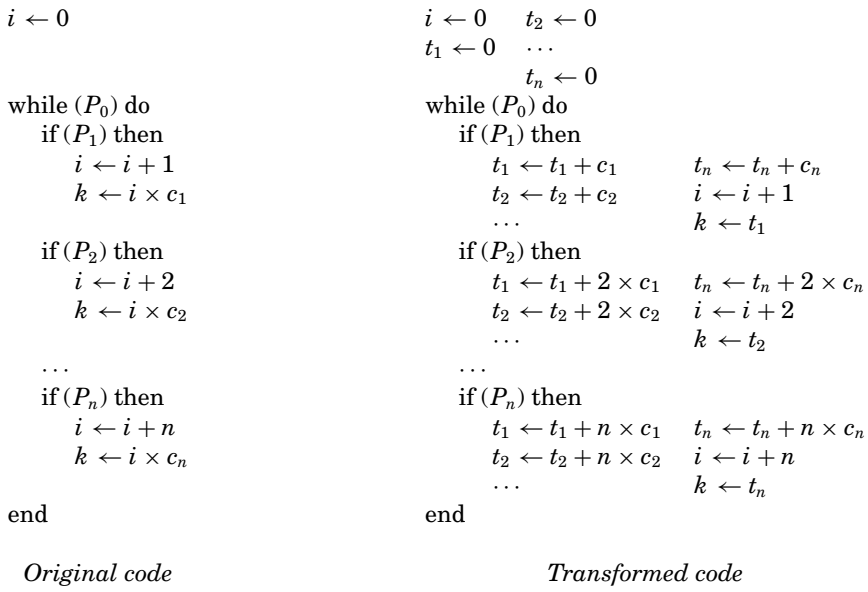

Fig. 9. A worst-case example.

generates this worst case behavior in the replacement step. It requires introduction of a quadratic number of updates. Note that this behavior is a function of the code being transformed, not any particular details of our algorithm. Any algorithm that performs strength reduction on this code will have this behavior. Experience with strength reduction suggests that this problem does not arise in practice. In fact, we have not seen an example with this behavior mentioned in the literature. Since the amount of work is proportional to the number of instructions inserted, any algorithm for strength reduction that reduces these cases will have the same, or worse, complexity.

4.6 Follow-up Transformations

OSR is intended to operate in a compiler that performs a suite of optimizations. To avoid duplicating functionality and to provide for a strong separation of concerns, our algorithm leaves much of the "cleaning up" to other well-known optimizations that should run after OSR.

OSR can introduce equal induction variables. Thus, the compiler needs a *global value numbering* algorithm to detect and remove common subexpressions. It must be a global technique that can handle values flowing along back edges in the CFG, as induction variables do. The partitioning technique of Alpern et al. will discover identical values [Alpern et al. 1988; Click and Cooper 1995], as will other global algorithms [Cooper and Simpson 1995; Briggs et al. 1997].

The SSA graph in Figure 8 contains a great deal of dead code. Many of the use-definition edges in the original SSA graph have been changed, producing "orphaned" nodes. OSR depends on a separate *dead code elimination* pass to remove these instructions [Cytron et al. 1991, Sect. 7.1].

Many of the copies introduced by OSR can be eliminated. For example, the COPY into $t3_0$ in Figure 8 can be eliminated if the load into $t4_0$ uses the value of osr_6 directly. Our compiler relies on the *copy coalescing* phase of a Briggs-Chaitin register allocator to accomplish this task [Chaitin et al. 1981; Briggs et al. 1994].

4.7 Comparing OSR with Allen-Cocke-Kennedy

OSR has several advantages over the classic ACK algorithm.

(1) OSR operates on SSA form. It uses the properties of SSA and data-structures built during SSA-construction to eliminate the passes needed by ACK to recognize the code's loop structure and to identify region constants. Using SSA also leads to a strategy that avoids instantiating the $\mathcal{IV}$ and $\mathcal{RC}$ sets. SSA-form also lets it iterate more efficiently over the code. For example, to find an induction variable, OSR examines only those operations that update the potential induction variable, where ACK visits all the operations in the loop.

(2) OSR reduces candidate instructions as they are encountered, rather than maintaining a worklist of candidates. This lets it directly reduce new induction variables as it creates them. ACK reduces new induction variables that it inserts by placing them on the worklist. It cannot, however, recognize when one reduction converts an existing variable into an induction variable. To find and reduce these variables, ACK must be invoked again on the entire SCR.[5]

(3) OSR produces results quite similar to those produced by ACK. The primary difference in the results lies in the placement of initializations. ACK creates a prolog block, or landing pad, for each loop to hold the initializations that it inserts. OSR inserts initializations after the definition of an operand that is closest to its use. This can place the operation in a less deeply nested location than the loop's landing pad, where it should execute fewer times.

(4) OSR removes a couple of subtle restrictions that apply to ACK. Since it largely ignores the CFG, OSR handles multiple-entry loops in a natural way. Similarly, ACK assumes that induction variables are defined in the loop's prolog block; OSR makes no similar assumption.

(5) OSR is easier to understand, to teach, and to implement than the ACK algorithm. It has been implemented in several research and production compilers, and used as an implementation exercise in a second-semester compiler course.

5. LINEAR FUNCTION TEST REPLACEMENT

After strength reduction, the transformed code often contains induction variables whose sole use is to govern control flow. In that case, linear function

[5]The data-flow methods also fail to reduce some newly-created induction variables. The analysis cannot address these variables since they are created after the analysis runs. Iterating the analysis and transformation should allow the data-flow methods to catch them.

test replacement may be able to convert them into dead code. The compiler should look for comparisons between an induction variable and a region constant. For example, the comparison "**if** $(i_2 \leq 100)$ **goto** L" in the ongoing example (see Figure 1) could be replaced with "**if** $(osr_8 \leq 396 + a)$ **goto** L". This transformation is called *linear function test replacement* (LFTR).

Previous methods would search the hash table for an expression containing the induction variable referenced in the comparison. In the example in Figures 4 and 8, a "chain" of reductions was applied to node i_2. If LFTR is to be effective, it must follow the entire chain quickly. To facilitate this process, `Reduce` can record the reductions it performs on each node in the SSA graph. Each reduction is represented by an edge from a node to its strength-reduced counterpart labeled with the opcode and the region constant of the reduction. In Figure 8, these edges are the dashed gray arrows. They would be labelled as follows:

$$i_1 \xrightarrow{-1} osr_0 \xrightarrow{\times 4} osr_3 \xrightarrow{+a} osr_6$$
$$i_2 \xrightarrow{-1} osr_2 \xrightarrow{\times 4} osr_5 \xrightarrow{+a} osr_8$$

When the compiler identifies a candidate for LFTR, it can traverse these edges, insert code to compute the new region constant for the test, and update the compare instruction. Our implementation uses two procedures to support this process:

`FollowEdges` follows the LFTR edges and returns the SSA name of the last one in the chain.

`ApplyEdges` applies the operations represented by the LFTR edges to a region constant and returns the SSA name of the result.

The `ApplyEdges` function can be easily implemented using the `Apply` function described in Section 4.3. For each LFTR candidate, it replaces the induction variable with the result of `FollowEdges`, and replaces the region constant with the result of `ApplyEdges`. Notice that LFTR renders the original induction variable dead; subsequent optimizations should remove the instructions used to compute it.

To transform the test $i_2 \leq 100$ in Figure 8, LFTR replaces i_2 with the result of `FollowEdges`, osr_8. Next, it replaces 100 with the result obtained from `ApplyEdges`, $(((100 - 1) \times 4) + a) = 396 + a$. The original induction variable, $\{i_1, i_2\}$, is no longer needed; it will be removed by optimizations performed later.

6. CONCLUSIONS

This paper presents OSR, a simple and elegant new algorithm for operator strength reduction. OSR produces results that are similar to those achieved by the Allen, Cocke, and Kennedy algorithm. It relies on prior optimizations and properties of the SSA graph to produce an algorithm that (1) is simple to understand and to implement, (2) avoids instantiating the sets of induction variables and region constants required by other algorithms, and (3) greatly

simplifies linear function test replacement. Rather than performing a separate analysis to discover the loop structure of the program, it relies on dominance information computed during the SSA construction. The result is an efficient algorithm that is easy to understand, easy to teach, and easy to implement.

ACKNOWLEDGMENTS

Our colleagues on the Massively Scalar Compiler Project at Rice have contributed to this work in many ways. Tim Harvey acted as a sounding board for many of the ideas presented here. Linda Torczon and Tim Harvey served as diligent proof readers of this text. The *Advanced Compiler Construction* class at Rice University implemented the algorithm for their class project. They provided us with a great deal of insight into how to present these ideas. Cliff Click provided the example shown in Figure 9. The anonymous referees and the editors of TOPLAS provided valuable suggestions, comments, and insights.

Vivek Sarkar of IBM provided support for Taylor Simpson through an IBM graduate fellowship; Reid Tatge of TI provided support for both Chris Vick and Keith Cooper. All of these people deserve our sincere thanks.

The staff of TOPLAS displayed infinite patience with our slow turnaround time. Any delay in publication is solely the fault of the first author.

REFERENCES

ALLEN, F. E. 1969. Program optimization. *Annual Review in Automatic Programming 5*, 239–308.

ALLEN, F. E., COCKE, J., AND KENNEDY, K. 1981. Reduction of operator strength. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, Englewood Cliffs, NJ, USA.

ALPERN, B., WEGMAN, M. N., AND ZADECK, F. K. 1988. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, San Diego, California, 1–11.

BERNSTEIN, R. 1986. Multiplication by integer constants. *Software—Practice and Experience 16*, 7 (July), 641–652.

BODIK, R., GUPTA, R., AND SOFFA, M. L. 1998. Complete removal of redundant computations. *SIGPLAN Notices 33*, 5 (May), 1–14. *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*.

BRIGGS, P. AND COOPER, K. D. 1994. Effective partial redundancy elimination. *SIGPLAN Notices 29*, 6 (June), 159–170.

BRIGGS, P., COOPER, K. D., HARVEY, T. J., AND SIMPSON, L. T. 1998. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience 28*, 8 (July), 859–881.

BRIGGS, P., COOPER, K. D., AND SIMPSON, L. T. 1997. Value numbering. *Software—Practice and Experience 27*, 6, 710–724.

BRIGGS, P., COOPER, K. D., AND TORCZON, L. 1994. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst. 16*, 3 (May), 428–455.

BRIGGS, P. AND HARVEY, T. J. 1994. Multiplication by integer constants. This is a "web", a literate programming document. See `http://softlib.rice.edu/MSCP`.

CAI, J. AND PAIGE, R. 1991. "Look Ma, no hashing, and no arrays neither". In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, Orlando, Florida, 143–154.

CHAITIN, G. J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., AND MARKSTEIN, P. W. 1981. Register allocation via coloring. *Computer Languages 6*, 47–57.

CHASE, D. R. 1988. Personal communication in the form of an unpublished report.

CHOI, J.-D., CYTRON, R., AND FERRANTE, J. 1991. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, Orlando, Florida, 55–66.

CLICK, C. AND COOPER, K. D. 1995. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst. 17*, 2 (Mar.), 181–196.

COCKE, J. AND KENNEDY, K. 1977. An algorithm for reduction of operator strength. *Communications of the ACM 20*, 11 (Nov.), 850–856.

COCKE, J. AND MARKSTEIN, P. 1980a. Measurement of program improvement algorithms. In *Proceedings of Information Processing 80*. North Holland Publishing Company, Tokyo, Japan.

COCKE, J. AND MARKSTEIN, P. 1980b. Strength reduction for division and modulo with application to a multilevel store. *IBM J. Res. Dev. 24*, 6, 692–694.

COCKE, J. AND SCHWARTZ, J. T. 1970. Programming languages and their compilers: Preliminary notes. Tech. rep., Courant Institute of Mathematical Sciences, New York University.

COOPER, K. D. AND SIMPSON, L. T. 1995. SCC-based value numbering. Tech. Rep. TR95636, Center for Research on Parallel Computation, Rice University. Oct.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct.), 451–490.

DHAMDHERE, D. M. 1979. On algorithms for operator strength reduction. *Communications of the ACM 22*, 5 (May), 311–312.

DHAMDHERE, D. M. 1989. A new algorithm for composite hoisting and strength reduction. *Int. J. Comput. Math. 27*, 1, 1–14.

DRECHSLER, K.-H. AND STADEL, M. P. 1993. A variation of Knoop, Rüthing, and Steffen's "lazy code motion". *SIGPLAN Notices 28*, 5 (May), 29–38.

EARLY, J. 1974. High level iterators and a method of automatically designing data structure representation. Tech. Rep. ERL-M416, Computer Science Division, University of California, Berkeley. Feb.

FONG, A. C. 1979. Automatic improvement of programs in very high level languages. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*. ACM, San Antonio, Texas, 21–28.

FONG, A. C. AND ULLMAN, J. D. 1976. Induction variables in very high level languages. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages*. ACM, Atlanta, Georgia, 104–112.

GRANLUND, T. 1995. Private communication with P. Briggs. Discussion of his work in building the routine `synth_mult` for the Gnu C Compiler.

GRANLUND, T. AND MONTGOMERY, P. L. 1994. Division by invariant integers using multiplication. *SIGPLAN Notices 29*, 6 (June), 61–72.

GUPTA, R., BERSON, D. A., AND FANG, J. Z. 1998. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*. IEEE Computer Society, Chicago, IL., USA, 230–239.

HAVLAK, P. 1997. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst. 19*, 4 (July), 557–567.

ISSAC, J. AND DHAMDHERE, D. M. 1980. A composite algorithm for strength reduction and code movement. *Int. J. Comput. Info. Sci. 9*, 3, 243–273.

KAM, J. B. AND ULLMAN, J. D. 1976. Global data flow analysis and iterative algorithms. *J. ACM 23*, 1 (Jan.), 158–171.

KENNEDY, K. 1973. Reduction in strength using hashed temporaries. SETL Newsletter 102, Courant Institute of Mathematical Sciences, New York University. Mar.

KENNEDY, K. 1978. Use-definition chains with applications. *Computer Languages 3*, 163–179.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1992. Lazy code motion. *SIGPLAN Notices 27*, 7 (July), 224–234.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1993. Lazy strength reduction. *J. Program. Lang. 1*, 1, 71–91.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1994. Optimal code motion: Theory and practice. *ACM Trans. Program. Lang. Syst. 16*, 4 (July), 1117–1155.

LENGAUER, T. AND TARJAN, R. E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 1 (July), 121–141.

LIU, Y. A. AND STOLLER, S. D. 1998. Loop optimization for aggregate array computations. In *IEEE 1998 International Conference on Computer Languages*. IEEE CS Press, Los Alamitos, CA, 262–271.

MARKSTEIN, P. W., MARKSTEIN, V., AND ZADECK, F. K. 1994. Reassociation and strength reduction. Chapter from an unpublished book, *Optimization in Compilers*.

MOREL, E. AND RENVOISE, C. 1979. Global optimization by suppression of partial redundancies. *Communications of the ACM 22*, 2 (Feb.), 96–103.

PAIGE, R. AND KOENIG, S. 1982. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst. 4*, 3 (July), 402–454.

PAIGE, R. AND SCHWARTZ, J. T. 1977. Reduction in strength of high level operations. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*. ACM, Los Angeles, California, 58–71.

SANTHANAM, V. 1992. Register reassociation in PA-RISC compilers. *Hewlett-Packard Journal 14*, 6 (June), 33–38.

SCARBOROUGH, R. G. AND KOLSKY, H. G. 1980. Improved optimization of FORTRAN object programs. *IBM J. Res. Dev. 24*, 6 (Nov.), 660–676.

SITES, R. L. 1979. The compilation of loop induction expressions. *ACM Trans. Program. Lang. Syst. 1*, 1 (July), 50–57.

TARJAN, R. E. 1972. Depth first search and linear graph algorithms. *SIAM J. Comput. 1*, 2 (June), 146–160.

TARJAN, R. E. 1974. Testing flow graph reducibility. *J. Comput. Syst. Sci. 9*, 355–365.

VICK, C. A. 1994. SSA based reduction of operator strength. M.S. dissertation, Rice University, Department of Computer Science.

WEGMAN, M. N. AND ZADECK, F. K. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13*, 2 (Apr.), 211–236.

WOLFE, M. 1992. Beyond induction variables. *SIGPLAN Notices 27*, 7 (July), 162–174.

WU, Y. 1995. Strength reduction of multiplications by integer constants. *SIGPLAN Notices 32*, 2 (Feb.), 42–48.

Received March 1998; revised July 2001; accepted July 2001