INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313-761-4700  800/521-0600
SSA-based reduction of operator strength

Vick, Christopher Allen, M.S.
Rice University, 1994
RICE UNIVERSITY

SSA-Based Reduction of Operator Strength

by

Christopher A. Vick

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Keith D. Cooper, Associate Professor, Chair
Computer Science

Linda Torczon, Faculty Fellow
Computer Science

James B. Sinclair, Associate Professor
Electrical and Computer Engineering

Houston, Texas

February, 1994
SSA-Based Reduction of Operator Strength

Christopher A. Vick

Abstract

Reduction of operator strength is a well-known code improvement technique. It seeks to improve compiler-generated code by replacing repeated multiplications with repeated additions. Opportunities often occur in array addressing expressions inside loops. Strength reduction is generally combined with linear function test replacement, an optimization which removes induction variables whose uses have been strength reduced away by rewriting loop exit tests. This can reduce both the number of instructions in loops which contain array references and their cost. Perhaps the best known technique for performing strength reduction and linear function test replacement is the eight step approach presented by Allen, Cocke, and Kennedy.

This work explores the potential benefits of using the Static Single Assignment (SSA) form of the program as the intermediate representation for performing strength reduction and linear function test replacement. It follows the basic form of the Allen, Cocke, and Kennedy technique, but many of the individual steps will be modified to take advantage of the special attributes of the SSA form wherever it improves either the asymptotic complexity of the operation or the precision of the information generated. It is intended that this work would be integrated into an optimizer comprised of a full suite of optimizations using the SSA representation.
Acknowledgments

Along the rocky road to my completion of this work a number of people have been instrumental in helping me to find my way, keeping my feet moving forward, and keeping me generally sane. I would like to thank all of them here. First I must thank my terrific advisor, Keith Cooper, who has acted as counselor, critic, teacher, helper, and friend. I cannot imagine having a more dedicated, knowledgeable, and nurturing advisor, teacher, and friend, and I have truly enjoyed every aspect of my work and play with him. I also owe a debt to Linda Torczon who helped me when Keith was unavailable and maintained quiet but needed support throughout my time at Rice. Cliff Click has been another key counselor, critic, and friend who kept me going when things were at their worst, taught me an amazing amount of computer science, and was always there to discuss a problem or idea. My fellow students Clara Jaramillo, Mike Paleczny, Jerry Roth, and Hariklia Tsalapatos provided much needed support, ideas, and friendship. Finally, and most importantly, I wish to thank my wife Marialice Vick. She encouraged me to return to school, supported me throughout all my trials, failures, and triumphs, and made all my efforts worthwhile. No accomplishment which I have achieved would have been possible without her. Thank you all so very much.
Chapter 1

Introduction

1.1 Motivation

Reduction of operator strength (strength reduction) is the replacement of costly (strong) instructions with less costly (cheaper) ones. An example of this is the replacement of the multiplication of an unsigned integer and a constant with a series of shifts and adds. This is a weak form of strength reduction [Ber86]. Many opportunities for strength reduction are introduced by the compiler itself. For example, to calculate addresses for array references, compilers generate integer multiplications that are invisible at the source level (see Figure 1.1). Strength reduction seeks to replace these instructions with an equivalent calculation based upon repeated additions applied to compiler generated temporary variables. Historically, integer multiplication has taken more clock cycles than integer addition, so replacing a multiply with an add has decreased the execution time of the program even when it did not reduce the number of instructions executed. If the calculation is inside one or more loops, its execution frequency is high, which magnifies the potential speedup from this optimization. After strength reduction, certain induction variables can be completely eliminated by shifting loop exit tests to use the temporary variables generated by strength reduction. This optimization is called linear function test replacement.

Even in machines where integer multiplication is fast, strength reduction can be a valuable optimization because it facilitates reassociation of loop invariant expressions [San92, CM89]. Reassociation reorders calculations (typically array address expressions). By exposing common induction variables, it can decrease register pressure and reduce the number of increment operations. It may also expose additional loop invariant code and more common subexpressions. Decreased register pressure can reduce spill code, eliminating costly memory references.
Allen, Cocke, and Kennedy presented the classic approach to strength reduction and linear function test replacement [ACK81]. Their algorithm can be broken down into eight basic steps:

1. Generate use-definition information.
2. Find loops and construct landing pads.
3. Find region constants.
4. Find induction variables.
5. Find candidate instructions.
6. Map symbolic expressions to temporary names.
7. Replace instructions.
8. Perform linear function test replacement.

This thesis explores the benefits of performing strength reduction on code in Static Single Assignment (SSA) form [CFR+89]. SSA is an intermediate representation that concisely relates control flow to the flow of values. The basic eight step form of the Allen, Cocke, Kennedy technique will be followed, but many of the individual steps will be modified to take advantage of the special attributes of the SSA form of
program representation. SSA helps in several ways; it improves either the asymptotic complexity of the operation, the precision of the information generated, the space requirements of the operation, or the complexity of the algorithm. Because it is intended that this work be integrated into an optimizer that includes a full suite of optimizations, the presence of other well known SSA-based optimizations will be assumed. Finally, because fully detailed, easily implementable descriptions of this optimization are difficult to find, we will attempt to specify our algorithm in a fashion that allows for a simple and straightforward translation to a full implementation.

1.2 Related Work

Strength reduction is an optimization with a long history. It first appeared in a somewhat specialized form in the IBM FORTRAN I compiler in the late 1950's [All81]. The more general version of strength reduction, which is a familiar portion of modern optimizing compilers, was developed during the 1960's and 1970's by Francis Allen, John Cocke, Ken Kennedy, and others, working individually and in concert [CK77, AC72, Ken73b, Ken73a, CS70]. Much of this work was unified and presented in “Reduction of Operator Strength” by Allen, Cocke, and Kennedy. This work established a general comprehensive algorithm for performing strength reduction as a loop-based optimization, using global data-flow analysis techniques and a sophisticated hash-table-based scheme for allocating temporary variables [ACK81]. In his Ph.D. thesis, David Chase proposed certain improvements and corrections to the Allen, Cocke, and Kennedy work. This corrected version became the basis for our work.

While current strength reduction techniques were being refined, a more general theory called “Finite Differencing” was also developing. This began with the work of Jay Earley on generalizing set iterators for a set-based programming language [Ear75]. Amelia Fong merged some of Earley's ideas with the work of Cocke and Kennedy to explicitly apply strength reduction techniques to set theoretic languages [Fon79]. Then, Paige and Koenig generalized this technique to a framework for program transformations on high level set theoretic languages [PK82]. While this work provides interesting theoretic backing for our techniques, it does not directly relate to our work. Instead, like Allen, Cocke, and Kennedy, we are focusing on loop-based optimizations for FORTRAN and similar imperative languages.
Another complete line of work that has been merged into our algorithm is described in the literature on the Static Single Assignment form of intermediate program representation, which itself grew out of Reif and Tarjan's work on birthpoints [RT82]. We directly rely upon the work of Wegman and Zadek for locating constant values [WZ91]. We also use the minimal SSA form developed by Cytron, Ferrante, Rosen, Wegman, and Zadek as our intermediate form [CFR+89]. We are further indebted to Michael Wolfe for his work on detecting induction variables for data dependence analysis, as this forms the basis for our improved technique for finding induction variables [Wol92]. We are also aware of a document by Markstein, Markstein, and Zadeck that presents an algorithm that combines strength reduction, loop invariant code motion, and reassociation. The document is in draft form; it provides careful detail in some areas and is less concrete in others. In the context of our compiler, we already have solutions to the problems other than strength reduction.

1.3 Static Single Assignment Form

Since our algorithm is based upon the Static Single Assignment form, it seems appropriate to briefly describe this representation and its properties (see Fig. 1.2) [CFR+89]. The SSA form of the program can be considered as a sparse representation of use-definition chains [Wol92]. The process of converting a program to SSA form begins with the Control Flow Graph (CFG), which is a directed graph \( G = (V, E, \text{Entry}, \text{Exit}) \), where \( V \) is the set of vertices representing basic blocks in the program, \( E \) is the set of edges representing the flow of control among the blocks, and \( \text{Entry} \) and \( \text{Exit} \) are special vertices representing the unique entry and exit points of the program. The process produces an SSA CFG, which is a directed graph with the same set of nodes and edges as the CFG, but with the basic blocks rewritten.

After a program has been converted to SSA form it exhibits two key properties:

1. There is exactly one reaching definition for each use of a value in the program.

2. At points in the CFG where the flow of control merges, special functions called \( \phi \)-functions are placed in order to merge the values for a given variable from each control flow path.

The details of converting a CFG to SSA form and the descriptions of various types of SSA forms are presented in "An Efficient Method of Computing Static Single Assignment Form" [CFR+89]. It gives an algorithm for generating the Minimal SSA
form. This is the form that we use throughout our work and will refer to as SSA (see Figure 1.2). This form of SSA has the above mentioned two properties along with a third property: \( \phi \)-functions are created only at merge points in the CFG where the merging block is not control dependent on the blocks containing the definitions to be merged. This avoids unnecessary \( \phi \)-functions in many locations where a naive translation to SSA would insert them. In particular, minimal SSA does not place \( \phi \)-functions at inner loop headers for values that are never defined in the inner loop, but are defined in the outer loop prior to the inner loop and used in the outer loop after the exit from the inner loop. For example, a loop index variable for an outer loop has no \( \phi \)-function in an inner loop unless it is modified inside the loop. This property is useful in analyzing induction variables [Wol92], a critical part of recognizing opportunities for strength reduction.

<table>
<thead>
<tr>
<th>Example intermediate code</th>
<th>Example SSA intermediate code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{sum} \leftarrow 0.0 )</td>
<td>( \text{sum}_0 \leftarrow 0.0 )</td>
</tr>
<tr>
<td>( i \leftarrow 1 )</td>
<td>( i_0 \leftarrow 1 )</td>
</tr>
<tr>
<td>if ((i &gt; 50)) goto (E)</td>
<td>if ((i_0 &gt; 50)) goto (E)</td>
</tr>
<tr>
<td>( L: )</td>
<td>( L: )</td>
</tr>
<tr>
<td>( t_1 \leftarrow i \times 50 ) ( L: )</td>
<td>( \text{sum}_1 \leftarrow \phi(\text{sum}_0, \text{sum}_2) )</td>
</tr>
<tr>
<td>( t_1 \leftarrow t_1 + 3 )</td>
<td>( i_1 \leftarrow \phi(i_0, i_2) )</td>
</tr>
<tr>
<td>( t_2 \leftarrow \text{id}(a + t_1) )</td>
<td>( t_{l0} \leftarrow i_1 \times 50 )</td>
</tr>
<tr>
<td>( \text{sum} \leftarrow \text{sum} + t_2 )</td>
<td>( t_{l1} \leftarrow t_{l0} + 3 )</td>
</tr>
<tr>
<td>( i \leftarrow i + 1 )</td>
<td>( t_2 \leftarrow \text{id}(a + t_{l1}) )</td>
</tr>
<tr>
<td>if ((i \leq 50)) goto (E)</td>
<td>if ((i_2 \leq 50)) goto (E)</td>
</tr>
<tr>
<td>( E: ) ( E: )</td>
<td>( E: )</td>
</tr>
</tbody>
</table>

**Figure 1.2** Example of SSA form

### 1.4 Roadmap

In Chapter 2 we will present an overview of our algorithm for strength reduction. Chapter 3 will present a more detailed explanation of our techniques for constructing our program representation and encapsulating program structural information. The
identification of region constants and instantiation of a set of induction variables will be given a detailed treatment in Chapter 4. Chapter 5 will set forth our methods for locating candidate instructions, performing instruction replacement, and handling linear function test replacement. We will conclude in Chapter 6 with a summary of our work and its benefits.
Chapter 2

Algorithm Overview

Our first task is to specify an algorithm for performing strength reduction on the SSA CFG. Because our work is largely an adaptation of and extension to the work of Allen, Cocke, and Kennedy, we present a high-level overview of the algorithm broken into sections that mirror the structure of their work.

1. Build the SSA Value Graph

   - We assume that the CFG representation of the program has been converted into SSA form prior to running our algorithm [CFR+89].
   - The SSA Value Graph (SVG) is a graph constructed as in Wolfe’s “Beyond Induction Variables”, but with provision for both forward and backward traversals [Wol92].

2. Find Loops and Construct Landing Pads

   - Use Tarjan’s flow graph reducibility algorithm to find loops and nesting information [Tar74].
   - Build a tree of loop nests and run Depth First Search (DFS) on the tree to assign both a DFS number and a largest descendent number for each loop.
   - Create new blocks prior to each loop header in the CFG to serve as landing pads for initialization instructions generated during strength reduction.
   - Tag each block with either its loop number or “-1” if it is not within a loop.

3. Find the Set of Region Constants (RC)

   - Assume that Sparse Conditional Constant Propagation (SCCP) has been run, accompanied by some form of Partial Redundancy Elimination (PRE) [WZ91, MR79].
   - Thus, the code has the properties that all constants are tagged and most loop invariant code has been hoisted upwards out of the loops.
- Identify values in $\mathcal{R}C$ by following the back-edges in the SVG to the definition points for the operands.
- Check for either a constant tag or a location in the CFG outside the loop.
- Determining that a value is defined outside the loop is done by:
  (a) Find the loop number of the current loop ($L_{cur}$).
  (b) Follow the SVG use-definition edge to the value definition point.
  (c) Identify the loop number in which the definition appears, if any ($L_{op}$), and the loop number of the largest numbered child loop of $L_{op}$ ($L_{op\text{child}}$).
  (d) If the instruction is not in any loop (i.e., $L_{op} = -1$), then it is in $\mathcal{R}C$.
  (e) If the instruction is in a loop, then if the loop number $L_{cur}$ is in the following range, it is not in $\mathcal{R}C$: $L_{op} \leq L_{cur} \leq L_{op\text{child}}$.
  (f) Else, the value is in $\mathcal{R}C$.
- Thus, the set $\mathcal{R}C$ is never specified explicitly for any loop and determination of $\mathcal{R}C$ status for any value is made during other processing.

4. Find the Set of Induction Variables ($\mathcal{TV}$)
- Revision of Wolfe’s approach, providing for the optimistic assumption of $\mathcal{TV}$ status [Wol92].
  (a) Run the Tarjan Strongly Connected Component (SCC) algorithm over the SVG to identify all strongly connected regions [Tar72].
  (b) Each instruction in a SCC is tagged with a unique value representing membership in that SCC.
  (c) Place each SCC on a worklist and tag as a member of $\mathcal{TV}$.
  (d) Remove a SCC from the worklist and test for $\mathcal{TV}$ status:
    i. All SCC’s are assumed to be in $\mathcal{TV}$ unless proven not to be.
    ii. A SCC is not in $\mathcal{TV}$ if:
       A. It contains instructions other than add, subtract, $\phi$-function, and copy.
       B. Any operand of a non-$\phi$-function instruction is not in $\mathcal{R}C$ or $\mathcal{TV}$ as shown by tags.
  (e) If a SCC is not an $\mathcal{TV}$, then follow SVG forward edges to uses of the SCC value.
  (f) If a use is in another SCC and the use is not in $\mathcal{R}C$ for that SCC, that SCC is put back on the worklist.
  (g) SCC tags are removed to indicate that SCC is not in $\mathcal{TV}$.

5. Find Set of Candidate Instructions (CAND)
• Make a pass over the set $I\cal V$ looking for instructions which use members of $I\cal V$ as operands and have the following forms:
  (a) $iv \times rc$ or $rc \times iv$ [CK77]
  (b) $iv \times iv$ [ACK81]
  (c) Others are possible [ACK81].
• Note: We do not instantiate the set of candidates because we can, and do, locate and reduce them during instruction replacement.

6. Map symbolic expressions to temporary names

• Use a hash table to map candidate expressions to temporary names.

• Hash function should adopt a canonical ordering of operands so that commutative operators are handled efficiently (i.e., $a \times b$ should hash to the same temporary name as $b \times a$).

7. Insert new instructions

• Candidate instructions are replaced with a copy of the temporary generated for that candidate.

• At each definition point for the induction variable used in the candidate, insert the pattern of instructions which update both the new temporary and the induction variable.

• The new instructions may contain new candidates, and if so, these are immediately reduced.

8. Linear Function Test Replacement

• If, after instruction replacement, there are no remaining uses of an induction variable except a loopback test, then that test can be modified to use one of the new temporary variables.

• After modifying the test, the induction variable and all of its defining instructions can be removed from the program.

The remaining chapters of this work will present detailed descriptions of these sections. Each description will contain a complete specification of that portion of the algorithm along with an analysis of the benefits offered by SSA and the time bound for computing that section of the new algorithm.
Chapter 3

Preliminaries

3.1 Build the SSA Value Graph

As in the Allen, Cocke, and Kennedy approach, we utilize use-definition information. However, for our algorithm, this information is embedded in the structure of the SVG. The exact representation used for the SVG allows us to embed both use-definition and definition-use information in the SVG in order to allow traversal of the graph in both the forward and the backward directions (see Figure 3.1). This simplifies the task of finding region constants and induction variables.

\[
\begin{align*}
\text{sum}_0 &\leftarrow 0.0 \\
\text{i}_0 &\leftarrow 1 \\
\text{if } (\text{i}_0 > 50) \text{ goto } E \\
L: & \quad \text{sum}_1 \leftarrow \phi(\text{sum}_0, \text{sum}_2) \\
& \quad \text{i}_1 \leftarrow \phi(\text{i}_0, \text{i}_2) \\
& \quad \text{t}_1^0 \leftarrow \text{i}_1 \times 50 \\
& \quad \text{t}_1^1 \leftarrow \text{t}_1^0 + 3 \\
& \quad \text{t}_2 \leftarrow \text{ld}(\text{a} + \text{t}_1^1) \\
& \quad \text{sum}_2 \leftarrow \text{sum}_1 + \text{t}_2 \\
& \quad \text{i}_2 \leftarrow \text{i}_1 + 1 \\
& \quad \text{if } (\text{i}_2 \leq 50) \text{ goto } L \\
E: & \quad \ldots
\end{align*}
\]

*Figure 3.1 Example of SSA code and associated SSA Value Graph*
The SVG is based upon the SSA form of the program. Because we assume an SSA-based optimizer, we start with the program represented as a CFG in SSA form, with basic blocks represented as lists of instructions. From the CFG we build the SVG by building an array of pointers where each value (expressed in our intermediate representation as a register number) is an index into the array of pointers to the instruction defining that value. Using this array, the edges are built by traversing the instructions in the CFG and constructing edges between each use and its corresponding definition in the program. The SVG differs from the CFG in that it contains edges which represent the flow of values, but does not contain edges representing the flow of control. The SVG is a convenient abstraction for locating potential induction variables [Wol92].

Since the program is already in SSA form, there is only one definition for each use in the program. As each instruction uses at most two values, the total number of edges in the SVG is proportional to at most two times the number of instructions in the CFG. Furthermore, since the nodes of the SVG are the individual instructions from the CFG, the entire SVG representation is proportional in size to the CFG. Thus the construction of the SVG is proportional in both time and space to the size of the CFG.

3.2 Find Loops

There are a number of techniques for finding loop structures in programs. For this work, we use a variation of Tarjan’s flow graph reducibility algorithm [Tar74]. The loop structure it generates helps to determine the set of region constants and to place landing pads, new empty blocks placed immediately prior to the loop header but after the loop guard. The landing pads serve as a location for initialization code generated by instruction replacement. Placing this code in a landing pad ensures that it is not inadvertently placed inside a different loop nest (if the original defining point was inside a different loop), and guarantees that this code only appears at one location in the program.

In order to determine the loop structure of the program, we traverse the basic blocks of the CFG using Depth First Search (DFS) (see Figure 3.2). For each edge, we check the preorder number of the source against the preorder number of the sink. If the edge goes from a block with a higher preorder number to a block with a lower preorder number:
Figure 3.2  Example Control Flow Graph and related Dominator Tree
1. the edge is a loop-back edge

2. the source is the exit of the loop

3. the sink is the header of a loop

All blocks in the CFG that are on a path from the loop header to the loop exit, where the path does not include the exit itself, are part of the loop. If any loop headers are included in the body of the loop, then their associated loops are also included. This can be posed as an instance of the disjoint set problem. Therefore, Tarjan's Union-Find technique can be used to merge inner loops with outer loops without repeatedly reinspecting the inner loops [Tar74]. As loops are found, a data structure is allocated for each loop; it contains a loop number field, a pointer to the block in the CFG that is the header of the loop, a set representing the blocks included in the loop, a list of pointers to loops that are inner loops, and the number of the largest numbered descendant in the loop tree. Each instruction inside a loop is then tagged with the loop number of the nearest enclosing loop. Thus, instructions in inner loops are tagged with the loop number of the inner loop, not the outer loop number.

In order to correctly generate the loop information, our algorithm must process loop bodies in strict inner to outer order. Tarjan's algorithm approximates this order by using a DFS pass to discover loop header blocks (sinks of back edges), which are placed on a stack as they are found. Since the headers are found in preorder, the headers will be processed in reverse preorder, which guarantees that inner loop headers will be seen before or at the same time as outer loop headers. However, because Tarjan's algorithm specifies no particular ordering for processing loops that share a header block, it does not guarantee strict inner to outer ordering. This causes no problems for his reduction algorithm, but necessitates a modification for our purposes. The modification involves generating an ordered list of back edges entering each block (smallest preorder source to largest) as the loop headers are found. This ordered list is used to guarantee correct inner to outer processing as loop bodies are discovered by specifying the order in which multiple back edges into a single header are processed.

This modification of Tarjan's algorithm directly produces a forest of loop nest trees because, during the processing of an outer loop body, the headers of all immediate inner loops will be encountered and can thus be added to a growing tree of loop nests. The special case where two loops share a header but neither set of blocks is a proper subset of the other is handled by our algorithm by arbitrarily assigning the
Loop A = \{1,2,3,4,5,6,7,8,9,10\}
Loop B = \{3,4,5,6,7,8,10\}
Loop C = \{3,4\}
Loop D = \{4,5,6,7\}
Loop E = \{7,8,10\}

Loop Block Listing

Figure 3.3 Example loop list and related loop tree

loop with the lowest preorder number exit block as the inner loop. This treatment is a conservative approximation of the true state of affairs. If the loop analysis were perfect, it would be seen that the values defined in the blocks which make up the intersection of the two loops are not region constants for either loop, but those values defined in (LoopA - LoopB) are region constants for LoopB and likewise for (LoopB - LoopA) and LoopA. Our treatment of this condition will give the proper behavior for the inner loop, but will treat all values defined by the inner loop as non-region constants for the outer loop. Although this analysis is not ideal, it does allow correct (though not optimal) handling of irreducible loops and is an improvement upon the standard technique of treating these as a single loop [ASU86].

The loop forest is converted to a tree by adding a special root node if necessary (see Figure 3.3). Then DFS is run over the tree, assigning each loop a new unique loop number as it is first encountered in the search and storing the number of the largest numbered descendent of that loop in the tree. This number is found by taking the maximum of several values, the return values of the recursive DFS calls for each node in the tree and the value of the node itself. This value becomes the return value for the DFS call at each node. Instructions in loops are retagged with their new loop number using a mapping generated during the renumbering process. This information allows us to determine which instructions are within any given loop nest for testing of region constant status.
3.3 Time Bounds

Construction of the pointer array and the edges of the SVG is linear in the size of the CFG, so the whole process of constructing the SVG is linear in the size of the CFG. The loop finding algorithm can be written in $O(E\alpha(V, E))$, which is very nearly linear in the size of the CFG [Tar74, Tar75]. The construction of the loop tree is linear in the number of loops in the CFG, which cannot be greater than linear in the size of the CFG. Similarly the DFS pass is linear in the size of the loop tree since it does constant work for each loop node and is thus also linear in the size of the CFG. Therefore, the time bound for all preliminary work in our algorithm is $O(E\alpha(V, E))$ for the CFG.
Chapter 4

Find the Sets of Region Constants and Induction Variables

4.1 Definitions

This chapter discusses new techniques for finding two types of values in a loop, region constants and induction variables.

Region constants are values that are constant throughout a certain region of the code, in this case, within a loop.

The fully general problem of finding all values that do not change in a particular section of code is undecidable in the presence of general control flow and either aliasing or pointer variables. Thus, all strength reduction algorithms use some conservative approximation for this set. Previous papers have provided sketchy descriptions of the approximation that they use. Our approach is simple and clear. We run Wegman and Zadeck's Sparse Conditional Constant Propagation (SCCP) algorithm and Partial Redundancy Elimination (PRE) and then consider any value that is either constant or has no definition within the loop being considered to be a region constant [WZ91, MR79]. With additional analysis of aliasing and memory references, the precision of the region constant sets could be extended, exposing additional induction variables and additional reduction candidates.

Induction variables are recursively defined using both region constants and induction variables.

An induction variable is a variable in a loop which is only defined using add, subtract, unary minus, copy, and $\phi$-function instructions whose operands are either region constants or induction variables.

While this definition is identical to that used by Allen, Cocke, and Kennedy [ACK81], we compute the set of induction variables in a different manner.
4.2 Region Constants

Previous algorithms have determined a set $\mathcal{RC}$ of region constants by relying on loop invariant code motion to identify the set $\mathcal{RC}$ for each loop. In a similar manner, this algorithm relies upon the prior execution of SCCP and PRE [WZ91, MR79]. Instead of relying on prior passes to generate and record a set of region constants for each loop or making a separate pass over all instructions in each loop to build a bit-vector representation of these sets, this algorithm relies on the ability to textually identify a constant and exploits the code shape which results from the prior passes. With the SVG representation, region constants can be found by following the edges in the SVG from each operand to its definition and then checking to see if that definition is either a constant or is outside the current loop and thus loop invariant. Testing for location outside the current loop is performed by comparing the loop number of the current instruction $L_{\text{cur}}$ with the loop number and the largest child number of its operand, $L_{\text{op}}$ and $L_{\text{opchild}}$ respectively. If $L_{\text{op}} \leq L_{\text{cur}} \leq L_{\text{opchild}}$, then the definition of that operand is inside the current loop and thus is not in $\mathcal{RC}$. Otherwise, the definition is outside the loop and therefore is in $\mathcal{RC}$.

Instead of iterating through the instructions of the loop to generate the set $\mathcal{RC}$ for each loop explicitly, we will determine whether a value is in $\mathcal{RC}$ during our processing to locate induction variables. This approach saves both time and space. The only region constants that are relevant for strength reduction appear as operands of instructions that define values for induction variables. Any other region constants are useless for strength reduction. In practice, finding them might generate a large amount of useless information. In addition, generating such sets would force us to store them and perform membership tests during our processing of potential induction variables; this is not necessary in our approach.

4.3 Induction Variables

In “Beyond Induction Variables,” Michael Wolfe introduced an SSA-based approach for finding various types of induction variables in the context of performing data dependence analysis [Wol92]. We have adopted his basic approach with some revisions designed to tailor the process to strength reduction. Our process is based upon the Tarjan Strongly Connected Component (SCC) algorithm [Tar72], which finds maximal SCC’s. This algorithm is run over the SVG to locate SCC’s. As each SCC is found, it is placed on a worklist and tagged as an induction variable. When all SCC’s are
on the worklist, an SCC is removed from the list and tested to see if it represents an induction variable. If it proves not to represent an induction variable, then each instruction in the SCC is revisited, following definitions in the SCC to their uses. Any such uses which appear in SCC’s are checked for region constant status. If the uses are not region constants, the SCC which contains such uses is placed on the worklist. This insures that the determination that a value is not an induction variable will be properly used in evaluating all other SCC’s. This, in turn, may result in another SCC being tagged as not in \(TV\). This worklist process is different from the process used by Wolfe in that it makes an optimistic assumption that each SCC in the SVG represents an induction variable until it is proven otherwise. This insures that the set \(TV\) will be as large as possible, giving the maximum opportunity for finding strength reduction candidates.

In order to determine whether or not a SCC represents an induction variable, we must find those SCC’s that are not induction variables. This is accomplished through a multi-part test. If a SCC contains instructions with opcodes other than add, subtract, copy, and \(\phi\)-node, then the SCC is not an induction variable and is tagged as not being in the set \(TV\). If a SCC has any operands to instructions other than \(\phi\)-functions that are either not region constants or are not in \(TV\), then the SCC does not represent an induction variable. In order to save processing time, each operand is first checked for status as a region constant. If the value is a region constant, then its status as an induction variable elsewhere is irrelevant. Only if the value is not a region constant is its status as an induction variable checked.

In addition to finding induction variables, our method allows us to classify them as either simple or complex in the same way that Wolfe does [Wol92]. Simple induction variables have a linear progression or “step”, whereas complex ones have a step which is polynomial or some other more complex function. Wolfe’s classification process relies upon an insight into the structure of the SVG. SSA form eliminates \(\phi\)-functions that are unnecessary. Thus, there are no \(\phi\)-functions in inner loops for variables of outer loops that are never redefined in the inner loop. Simple induction variables will only have a single \(\phi\)-function at a loop header contained within their SCC. The form of a SCC with a single loop header \(\phi\)-function must represent a simple loop in the CFG because each back edge in the SVG must connect to a \(\phi\)-function, \(\phi\)-functions only appear at points that merge the flow of control in the CFG, and only simple loops have a single back edge merge point. Likewise, more complex forms of induction
variables can be identified by the presence of more than one loop header $\phi$-function in the SCC and by the shape of the SVG around those extra $\phi$-functions.

4.4 Time Bound

The entire process of finding region constants and induction variables is linear in the size of the SVG. The Tarjan SCC algorithm is $O(V + E)$ [Tar72] and the worklist adds $O(E)$ time. The worklist is limited by the fact that each SCC can fall out of $\mathcal{T}V$ at most once. Thus, each SCC can be processed on the worklist at most twice because once it is processed and found to be in $\mathcal{T}V$, it will get back on the worklist only if a value upon which it relied to be in $\mathcal{T}V$ has fallen out of $\mathcal{T}V$. Once this happens, the SCC will also fall out of $\mathcal{T}V$ when it is processed for the second time and will thereafter never again be placed upon the worklist. Since $L_{op}$ and $L_{opchild}$ are stored in the loop data structure for each loop, finding each of these values is a constant time lookup of the loop number of the definition point and then the comparison of values in the relevant loop data structure. Since membership in $\mathcal{RC}$ is a constant time operation, the determination of whether a value is in $\mathcal{T}V$ is also a constant time lookup. The instructions in each SCC are tagged as being in $\mathcal{T}V$ when they are put on the worklist and tagged as not being in $\mathcal{T}V$ when they are determined not to be induction variables. All other variables tagged as not in $\mathcal{T}V$ (all instructions in the SVG have a SCC pointer, which is NULL for those instructions that are not in a SCC, and each SCC has a flag to indicate its status as an induction variable). Thus, there is at most $O(V)$ time spent walking the instructions and checking operands for all the SCC’s because each check is a constant time lookup and there are a small constant number of operands for each instruction (generally two). The other major time component of finding induction variables is the time required to iterate over the instructions in each SCC which has been removed from the set $\mathcal{T}V$. During this walk the forward edges to uses of the values defined in the SCC are traversed in order to locate further SCC’s to place back on the worklist and reevaluate their status as an induction variable. Since each SCC can be determined to not be in $\mathcal{T}V$ only once, this involves at most walking $O(E)$ edges total for all SCC’s. Since the SCC portion is $O(V + E)$ and the determination of which SCC’s are in $\mathcal{T}V$ is also $O(V + E)$, the entire process is $O(V + E)$ which is linear in the size of the SVG.
Chapter 5

Loop Processing and Instruction Replacement

5.1 Find Sets of Candidate Instructions and Critical Points

Once induction variables have been identified it becomes possible to identify the set, CAND, of instructions which can be strength reduced. These instructions are called candidates, and their definition depends upon exactly what types of computations the compiler writer wishes to reduce. A large class of computations can be strength reduced; Allen, Cocke, and Kennedy provide a catalog of some of the more complex calculations [ACK81]. In this work we have chosen to limit ourselves to those instructions in a loop which take the form of a multiply of some combination of induction variables and region constants. This simplifies our presentation. The more complex reductions are performed in the same manner, using their own particular instruction substitution patterns [ACK81]; their inclusion in a strength reducer based upon this work is straightforward. The fast way to locate candidate instructions is to make a pass over the set $\mathcal{IV}$, and check each instruction that uses a member of $\mathcal{IV}$ (follow SSA edges to uses) to see if it is a multiply and its other argument is in either $\mathcal{IV}$ or $\mathcal{RC}$. Those instructions which meet the definition are candidates and can be directly reduced.

Allen, Cocke, and Kennedy work from a set of critical points. The set of critical points is defined as the set CAND unioned with the set of definition points for the elements in $\mathcal{IV}$. Using SSA form, we can instead work from the SCC's, following edges in the SVG back to all definition points. This allows us to avoid explicitly finding and representing the set of critical points. These facts permit us to further simplify our algorithm by finding candidate instructions on the fly as we perform instruction replacement. Since all instructions must be placed at definition points specified by the instructions that make up an induction variable’s SCC, along with the definition points for all values flowing into $\phi$-nodes in the SCC (which are accessible due to the Use-Def chains embedded in the SVG), a single pass over the set of induction variables allows us to find each candidate instruction, and immediately reduce it as it is found.
5.2 Insert New Instructions

A mapping of expressions involving induction variables to temporary names is required, and can be implemented via a hash table which hashes on the operands and the opcode of the original expression. This creates a new name for each entry in the table which is then used as the name of the temporary variable [CK77]. This process can be made more efficient for the simple case of reducing multiplications and additions by insuring that the hashing function is not sensitive to the ordering of the operands (i.e., $a \times b$ hashes to the same name as $b \times a$). This table is used during instruction replacement, and need contain only those expressions which are selected for reduction.

The technique for replacing instructions based upon induction variables with instructions based upon the newly generated temporaries is described in detail in "Reduction of Operator Strength" [ACK81]. That technique searches all of the instructions in each loop for candidate instructions, and instantiates the set of candidate instructions as a worklist. For each candidate, Allen, Cocke and Kennedy replace that instruction with a copy of a newly generated temporary value, and then insert defining instructions for the new temporary at each critical point in the loop. Our technique is a minor simplification of that technique. A pass over the list of induction variables is made, checking each use of a value defined in the induction variable’s SCC. As each candidate is located, the instruction is hashed into the temporary table and a new temporary name is generated. We then perform the following two step process:

1. The candidate instruction is replaced with a copy of the temporary created for that candidate.

2. Each definition of the induction variable being processed is replaced with a sequence of instructions that define both the induction variable and the new temporary variable (see Figure 5.1).

The second step may create new candidate instructions. These are immediately reduced prior to returning to the search for other candidate instructions.

The properties of SSA let us simplify and reorganize the process used by Allen, Cocke, and Kennedy. Rather than use a worklist of candidate instructions, we can work from the list of SCC’s constructed earlier. The algorithm examines the SCC’s in some order. It traverses each SCC and follows the SSA edges from each definition to its uses. At each use, it uses the constant time test described in section 5.1 to determine if the use is a candidate for reduction.
For each candidate instruction $C$, the compiler applies the following steps. (Assume that the compiler is processing the SCC for $i$ and that $C$ has the form $x \leftarrow i \times j$.)

1. Replace $C$ with $x \leftarrow t_{ij}$.

2. $C$ has a unique definition for $i$. If that definition is not followed by an update of $t_{ij}$, then for each definition in the SCC and each definition that reaches a $\phi$-node in the SCC:

   (a) If the definition is not the $\phi$-node at the head of the loop, replace the definition with the update sequence shown in Table 5.1.

   (b) If the definition is the $\phi$-node at the head of the loop, then, for each definition point that is post-dominated by the $\phi$-node, insert the update sequence shown in Table 5.1 in the landing pad for that loop.

   (c) Recurse on any new candidates introduced by the replacement.

The inserted instructions can create new candidates. This is possible on each instruction marked with maybe in Table 5.1. If both operands are region constants, the expression is subject to either constant folding or loop invariant code motion. In either case, the instruction simplifies to an assignment. If one or both operands are induction variables, the new instruction is itself a candidate.

Step 2 checks for an existing update to $t_{ij}$. This prevents the compiler from inserting duplicate code and from repeating work. It also ensures termination. This step also checks for the $\phi$-node at the head of the loop, and handles it in a special way. Those definitions which are post-dominated by the $\phi$-node are placed in the loop's landing pad rather than being located at the definition points for the $\phi$-node in order to prevent the insertion of code into a different loop nest, and to simplify the process of inserting instructions outside the loop currently being processed.
5.3 Linear Function Test Replacement

After strength reduction, the code may contain induction variables whose sole use is to govern control flow. If the induction variable has the following properties:

1. all remaining uses are either comparisons that control branches or self-updates, and

2. the compiler has created a temporary variable that is a linear function of the original induction variable,

then it can apply a transformation called linear function test replacement to convert the remaining uses into dead code.

The compiler looks for uses of the form

\[ \text{if } i \leq c \text{ then go to loop} \]

where \( c \) is a region constant. If the compiler has inserted code to initialize and update \( t_{ij} \) and \( j \) is a region constant, this test can be replaced by

\[ \text{if } t_{ij} \leq c \times j \text{ then go to loop} \]
Note that $c \times j$ is a region constant and susceptible to further optimization — either constant folding or loop invariant code motion. After this transformation, dead code elimination should remove the extraneous initialization and update code for $i$.

To perform linear function test replacement, the compiler should examine uses during the replacement phase. As it processes each SCC, it can build a short list of control-flow tests. As it reduces candidates, it can record the first linear function that it creates. At the end of that SCC, it can re-visit the tests and transform them to use the newly created induction variable.

5.4 Time Bound

The check for CAND status can be done in constant time by following the backwards SVG edges. Thus, the set CAND can be found in linear time. Since the number of new instructions inserted is proportional to the number of candidate instructions times the number of induction variables, the total number of new instructions is potentially quadratic in the size of the intermediate representation. The quadratic bound is generated by a pathologic case where a single loop contains $O(n)$ conditionals with distinct, non-constant predicates. Each of these conditionals contains an update to the induction variable and a candidate instruction which uses that update. This forces the insertion of $O(n)$ instructions at each of $O(n)$ definition points. This result is a general one which applies to any general algorithm for operator strength reduction. Thus, the total work of finding candidate instructions and performing instruction replacement is quadratic in the worst case, although extensive prior experience with strength reduction indicates that it is linear in practice. The improvement in our work over previous work is a constant factor caused by our not having to iterate over every instruction in a loop to find candidate instructions, and our ability to locate all necessary definition points in linear time.
Chapter 6

Conclusions

The Static Single Assignment form of intermediate program representation provides a strong basis for performing strength reduction. Its use facilitates the formulation of a near linear algorithm for this optimization. Using SSA, we have developed simple and straightforward methods for finding region constants and induction variables. These methods are both fast and space efficient. SSA also facilitates the location of candidate instructions and critical points, allowing the construction of a strength reduction algorithm which is very similar to the Allen, Cocke, and Kennedy approach but with distinct benefits in space utilization, time complexity, and coding complexity. We have attempted to specify our algorithm in sufficient detail to permit a straightforward translation from our description to an implementation in order to enhance the utility of this work.

| Summary |
|------------------|------------------|------------------|------------------|
| **Step**              | **Allen, Cocke, Kennedy** | **New method** | **Comments** |
| 1. Construct Representation | CFG and def-use information | Minimal SSA and SVG | Helps later |
| 2. Find Loops and Insert Pads | Given as input | Flow graph reducibility and loop tree | |
| 3. Find Region Constants | Given as input | SCCP, PRE, and O(1) dynamic test | Don’t instantiate sets |
| 4. Find Induction Variables | Linear search over loop instructions | SCCs in SVG, optimistic algorithm | Optimism ⇒ more ZV’s SSA ⇒ single pass |
| 5. Find Candidates and Critical Points | Linear search over loop instructions; follow use-def chains | Linear search on SCC, candidates and critical points not instantiated | SSA ⇒ no critical points; inspect fewer instructions |
| 6. Map Names | Hash Table | Hash Table | |
| 7. Replace Instructions | Iterate over candidate worklist and insert at critical points | Iterate over and insert in SCCs | Don’t instantiate candidate set; inspect fewer instructions |
| 8. Linear Function Test Replacement | Follow def-use chains to find used values | Follow SVG edges to find used values | |
Bibliography


