Implementing Partial-Redundancy Elimination in SSA form
15-745 Project Report

Nick Hopper Dan Rosewater
{hopper,drose}@cs.cmu.edu

May 2, 2001

1 Introduction

Static Single Assignment (SSA) form is a well-known program representation in which every location has at most one static assignment [3]. SSA is widely-used by optimizing compilers to perform global analyses because of its accurate and concise representation of use-def relationships among program variables. Until recently, though SSA had been used to implement a wide variety of optimizations, it had not been used to implement Partial Redundancy Elimination, an important global optimization [2].

Partial Redundancy Elimination (PRE) is a global optimization which optimizes expressions which are redundant along some paths. Because it is powerful enough to subsume several previous local and global optimizations, including common subexpression elimination and loop invariant code motion, it has become an important component in many global optimizers [2]. It is frequently implemented as a dataflow problem, sometimes with bidirectional components.

Chow *et al* [2] describe an algorithm for performing PRE on a program in SSA form. This sparse algorithm computes PRE for a single expression E by introducing a hypothetical temporary h for E, creating an SSA graph for h and performing sparse dataflow analyses on this graph to construct a graph which eliminates partial redundancies where possible. In this report, we describe our implementation and preliminary empirical evaluation of this algorithm using the MachSUIF [5] compiler framework's new `ssa` library.

2 Previous Work

Chow *et al* have implemented their algorithm in the Silicon Graphics MIPSpro compiler and cite details concerning the running time of their PRE algorithm on the SPECint95 and SPECfp95 benchmark suites [2]. Presumably various other optimizing compilers which use SSA form have also done so. On the other hand we are aware of no effort to implement this optimization within the MachSUIF framework, and we are not aware of any empirical evaluation of the effectiveness of the optimization for any standard benchmark suite.

3 Implementation

The MachSUIF `ssa` module implements conversion to SSA form using the algorithm described by Briggs *et al* [1]. We functioned as beta-testers for this module.

Our implementation functions as a single MachSUIF pass operating on a CFG. After converting the CFG to SSA form, the SSA graph is analyzed to find instructions containing arithmetic expressions with the potential for partial redundancy. After all the "interesting" expressions are discovered, the procedure is then analyzed via multiple calls to a class hierarchy constructed to perform single-expression PRE. The class `ExprPRE` performs the six steps of the algorithm as described by [2]: Φ-insertion, Renaming, DownSafety, WillBeAvail, Finalize, and Codemotion. We implemented the Renaming pass as described in the **Delayed Renaming** section of [2], since it is more efficient for sparse representations because it does not require renaming stacks or traversal of the entire graph.

There is also a supporting hierarchy of classes representing occurrences of the hypothetical h. The `Occ` class is the base class for all occurrences; it contains all of the flags which the algorithm sets and references on a per-occurrence basis. The subclasses `RealOcc`, `PHINode`, and `PhiOp` represent occurrences associated with an actual instance of the expression, a Φ node for h, and an argument to a Φ node, respectively. The `PhiOp` is associated with the exit of the corresponding predecessor block, as required by the algorithm in [2]. These nodes support the basic operations on occurrences required by the algorithm.

Our pass was based on the beta version of the `ssa` conversion code, so naturally we expect to encounter a few bugs. The original form of the conversion code did not contain any methods for inserting or deleting new instructions and variables into the SSA form, so we contacted the author and described the interface we desired. He was gracious enough to provide a close approximation to this interface. However, we uncovered several additional instances where we required access to protected members of `ssa`'s `SsaForm` class. We also discovered a bug in SSA conversion where in some rare instances null is used as a ϕ operand, which should not happen. We contacted the author about this, but did not receive a response in time to include it in this report. We overcame this problem by ignoring null ϕ operands in the Φ-insertion step, where the problem occurred.

Most importantly, the `ssa` conversion code currently does not support persistent storage in SSA form, meaning that every pass that requires SSA must convert to SSA, perform the optimization, and convert back to CFG. Clearly this is suboptimal; for our pass, we perform many extra steps to ensure that we can insert new variable as needed to maintain SSA form, yet the SSA form is destroyed at the the end of the procedure analysis. The full benefit of our pass must await the ability to save to disk in `ssa`.

4 Results

In this section we give the results of two sets of experiments intended to measure the functionality of our SSAPRE module. The first set, the "Sanity Check" consist of two simple procedures with obvious partial redundancies that can be eliminated. Happily, our SSAPRE module eliminates both redundancies. The second set consist of some of the SPEC95 benchmark programs. Here we report the number of redundant expressions removed for each program.

4.1 Sanity Check results

We define two "sanity check" test cases: **loop** contains a short loop with a simple loop-invariant expression, which can be hoisted above the loop, as illustrated in Figure 1. **branch** redundantly evaluates an expression along one edge of a branch and not the other; it can also be eliminated. Figure 2 shows this case.

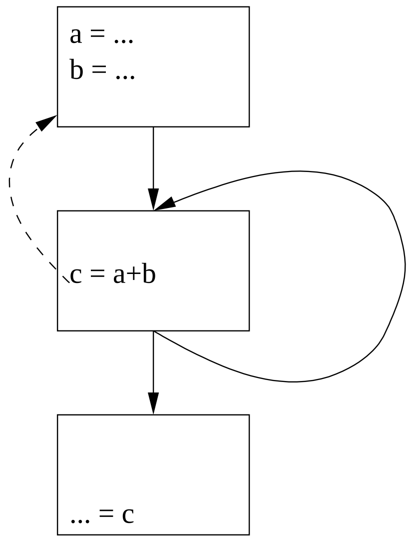

Figure 1: **loop**: The invariant computation `c = a + b` can be hoisted from the loop, as the dotted arrow indicates

`Name`	`go`‡	`m88ksim`‡	`gcc`†	`compress`	`li`	`ijpeg`‡	`perl`†	`vortex`†
Inserted	6	15	0	0	0	25	2	7
Removed	52	6	7	1	0	48	4	2
Expr considered	7420	9674	3524	258	1742	13522	4079	12747

Table 1: Results of applying PRE to SPECint95 benchmarks. †at least one source file did not parse using scc. ‡segfault in `do_ssapre` for at least one source file.

Examining the relevant code sections from the non-PRE and PRE intermediate representations of these examples shows that the redundancies were indeed eliminated. See Figures 3 and 4 for details. These results function as a sanity check of our module because it has the expected behavior.

We also incorporated the code from **loop** into a working C program that performed 100,000 iterations of redundant computation, then returned the result c. The assembly generated after performing PRE, DCE, register allocation, DCE again, and peephole optimization is shown in figure 5. We expected that, after performing PRE, subsequent optimization steps (namely, dead code elimination) would remove the loop which does not contribute to the result. However, we discovered that this is not the case. The store location of the move introduced in PRE is register `$0`, the return value for the function. Although this register could be coalesced with `$3` to eliminate the need for this move, `do_raga` did not do so. Clearly, a major benefit of PRE (elimination of unnecessary loops) awaits an optimization pass that can discover such constructs.

4.2 SPEC results

We applied our pass to the 8 programs in the SPECint95 benchmark suite. The results are recorded in Table 1. We discovered the presence of memory-related bugs which prevented successful compilation of several of the source files. Time constraints did not allow us to fix these bugs. We also found that the scc front-end to SUIF was not able to parse several of the source files. For the remaining source files, very few redundant expressions were found. While it is possible that

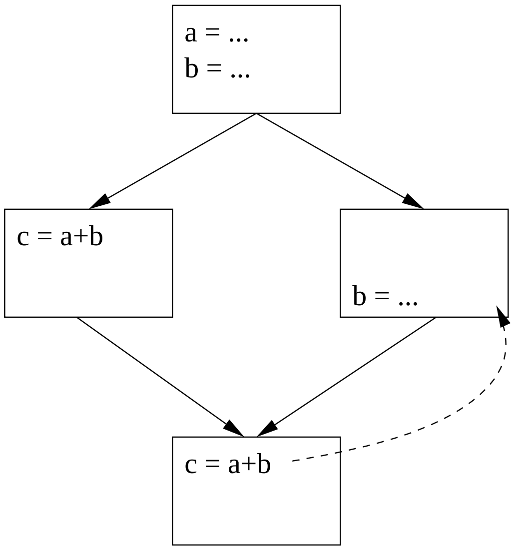

Figure 2: **branch**: The redundant computation `c = a + b` in the join block can be hoisted to its predecessor, as the dotted arrow indicates

```
foo:
        mov     $16,foo.a
        mov     $17,foo.b

        bne     $vr0,foo.L1

        addl    foo.a,foo.b,foo.c
        br      foo.__done2
foo.L1:

        ldil    foo.c,14
foo.__done2:
        addl    foo.a,foo.b,foo.c

        mov     foo.c,$0
        ret     ($26),1
```

```
foo:
        mov     $16,foo.a
        mov     $17,foo.b

        bne     $vr0,foo.L1

        addl    foo.a,foo.b,$vr25
        br      foo.__done2
foo.L1:

        ldil    foo.c,14
        addl    foo.b,foo.a,$vr25
foo.__done2:
        mov     $vr25,$0

        ret     ($26),1
```

Figure 3: The relevant code from **branch** before (left) and after (right) PRE

this is the result of a bug, we think it more likely that the programs do not contain much partial redundancy.

A possible explanation for this failure is the presence of "human PRE." It seems likely that most coders do things like hoist invariant code out of loops by hand, because they do not know or expect that the compiler will be able to automatically. Thus it is hard to assess the benefits of PRE in this situation.

5 Future Work

Due to certain deficiencies in the interface to the current MachSUIF ssa module, our module is tightly coupled to the implementation of that module. Perhaps an adapter class can be constructed which will allow looser coupling between the modules.

Chow *et al.* mention the possibility of accumulating a list of new expressions generated as a result of PRE, giving rise to elimination of redundant higher-order expressions which would not be possible in a dataflow-based PRE formulation. We have not implemented this feature but it could be done with another week's work. Along related lines, many compilers implement a notion

```
foo:
        mov     $16,foo.a
        mov     $17,foo.b

        ldil    foo.i,0
foo.L3:
        addl    foo.a,foo.b,foo.c
        ldil    $vr5,1
        addl    foo.i,$vr5,foo.i
        ldil    $vr9,10
        cmple   $vr9,foo.i,$vr8
        cmpeq   $vr8,0,$vr7
        bne     $vr7,foo.L3
foo.L2:
        mov     foo.c,$0
        ret     ($26),1
```

```
foo:
        mov     $16,foo.a
        mov     $17,foo.b

        ldil    foo.i,0
        addl    foo.b,foo.a,$vr29
foo.L3:
        mov     $vr29,foo.c
        ldil    $vr5,1
        addl    foo.i,$vr5,foo.i
        ldil    $vr9,10
        cmple   $vr9,foo.i,$vr8
        cmpeq   $vr8,0,$vr7
        bne     $vr7,foo.L3
foo.L2:
        mov     foo.c,$0
        ret     ($26),1
```

Figure 4: The relevant code from **loop** before(left) and after (right) PRE

```
        ...
        ldiq    $1,0
        addl    $3,$2,$3
main.L3:
        mov     $3,$0     <====
        ldiq    $4,1
        addq    $1,$4,$1
        ldiq    $4,1000000
        cmple   $4,$1,$4
        cmpeq   $4,0,$4
        bne     $4,main.L3
main.L2:
        lda     $sp,48($sp)
        ret     ($26),1
```

Figure 5: **loop** as part of a working C program. The move introduced by PRE (indicated by the arrow) impedes the removal of the loop.

of expression equivalence which is strong enough to allow PRE to subsume induction variable strength reduction. We believe it would be relatively straightforward to implement such a notion in our system. Going further afield, given good aliasing information, we could use our module to do PRE on loads and stores as in the later work of Chow *et al* [4]. We expect that this approach would offer more substantial speedups than expression PRE alone.

References

[1] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software: Practice and Experience*, 28(8):859–881, July 1998.

[2] F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on ssa form. In *Proc. of SIGPLAN '97 Conf. on Programming Language Design and Implementation (PLDI)*, pages 273–286, 1997.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] R. Lo, F. Chow, R. Kennedy, S. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proc. of SIGPLAN '98 Conf. on Programming Language Design and Implementation (PLDI)*, 1998.

[5] M.D. Smith. Extending suif for machine-dependent optimizations. In *Proc. First SUIF Compiler Workshop*, pages 14–25, Stanford, CA, January 1996.