Using SSA Form in a Code Optimizer

Carl McConnell
Ralph E. Johnson
University of Illinois at Urbana-Champaign*†

Abstract

We have constructed a simple yet powerful code optimizer that uses SSA form in concert with table-driven techniques for peephole optimization and code generation. This paper describes the problems with data dependences and 2-address instructions we encountered in combining these two techniques, and how we solved them.

1 Introduction

Static single assignment (SSA) form represents data dependences elegantly and provides a basis for powerful optimizations [2] [6] [20] [21]. Table-driven techniques for peephole optimization and code generation [1] [10] [11] [12] [13] [14] are straightforward and effective. It is natural to want to use both together in a code optimizer. However, several problems arise in doing this:

- SSA form does not capture the anti dependences [18] resulting from memory references.
- SSA form requires 3-address instructions, while table-driven techniques for code generation require the use of templates with the same form as the target machine, which is often 2-address.

*Authors' address: Department of Computer Science, 1304 W. Springfield, Urbana, IL 61801 USA. *E-mail*: `johnson@cs.uiuc.edu, mcconnel@cs.uiuc.edu`. *Phone*: (217) 244-0093.

†This research was supported by the National Science Foundation under grant CCR-8715752 and by a gift from Tektronix.

This paper shows how to make SSA form compatible with both anti dependences and table-driven code generators, and describes how this was done in the RTL System [17]. The RTL System is a language- and machine-independent compiler back-end. It performs optimizations like code motion, common subexpression elimination, and peephole optimization, as well as minimization of pipeline delays and filling of branch delay slots for RISC machines.

The RTL System is based on ideas developed by Davidson and Fraser [7] [8] [9] and implemented in their code generator PO. One of these ideas was that an intermediate language for a compiler should be the intersection of machine languages rather than the union, and that this language should be used not only for programs but also to describe the target machine architecture. Figure 1 shows a program written in the register transfer language of the RTL System. A program in this language can have an arbitrary number of logical registers, which are mapped to physical registers during register assignment.

The RTL System has the same three phases as PO. PO performs local optimizations in the first phase, while the RTL System performs global optimizations using algorithms based on SSA form. However, using SSA form in the first phase created problems in the second, which performs table-driven peephole optimization and code generation using the same greedy "combining" algorithm as PO. Section 4 describes how we solved these problems. Unlike the discussion of anti dependences in Section 3, which should be of general interest, the material in Section 4 is important primarily to those using a PO-style code selection algorithm, though similar problems would probably arise with other table-driven code selection algorithms. GCC, a widely-used C compiler whose source is freely available, employs PO's code selection algorithm, so improvements to it are useful even though newer and faster algorithms exist.

The third phase differs in some minor respects between PO and the RTL System: the RTL System generates machine language, while PO generates assembly language; and the RTL System does global register assignment using graph coloring, while PO just does local assignment. Since these differences were not caused by the use of SSA form, they are not discussed in this paper.

Compute the sum of 10 integers
starting at the memory location pointed to by r1,
and put the result in the memory location pointed to by r5.

```
r1 ← 1000.          Initialize pointer.
r2 ← 0.             Initialize sum to 0.
r3 ← 0.             Initialize counter to 0.

L1:
r3 < 10 ↑ L2.       If counter is less than 10, jump to L2.
r4 ← *r1.           Fetch integer pointed to ...
r2 ← r2 + r4.       ... and add it to sum.
r1 ← r1 + 4.        Increment pointer.
r3 ← r3 + 1.        Increment counter.
↑ L1.               Jump to L1.

L2:
*r5 ← r2.           Store sum.
@.                  Return.
```

Figure 1: A simple register transfer program.

2 Static Single Assignment Form

Global optimizations require information about the global data flow properties of a program: which assignments create the values used by a given assignment? This question is easy to answer if each variable is assigned a value exactly once, and this is guaranteed by SSA form. The RTL System uses SSA form in algorithms for common subexpression elimination [2], code motion out of loops [20], and other optimizations.

"Single assignment" languages are functional languages in which each variable is assigned a value exactly once at run-time, not merely in the program text. In contrast, any program can be put into static single assignment form; it is merely an alternative representation.

It is simple to rename variables in a basic block so that there is only one assignment to (i.e., definition of) each variable. However, this is not possible in the presence of conditionals or loops, since a given point in the program may be reached by conflicting definitions. In SSA form, the first node reached by a set of conflicting definitions starts with a *ϕ-assignment* that merges them. A ϕ-assignment has the form $U \leftarrow \phi(V, W, \ldots)$, where $U, V, W, \ldots$ are variables, and the number of operands $V, W, \ldots$ is the number of predecessors of the flow node in which the ϕ-assignment appears. If control reaches the ϕ-assignment through the j-th predecessor, then U is assigned the value of the j-th operand. Each execution of a ϕ-assignment uses only one of the operands, but which one depends on how control reached the flow node containing the ϕ-assignment.

Other investigators have used SSA form at the source level, applying it to variables, arrays, and pointers. Since we are working at the intermediate level, we make the obvious translation and instead apply SSA form to registers and "arrays" representing machine memory; pointers are of no concern, since at the intermediate level they become memory references. Figure 2 shows the register transfer program of Figure 1 after it has been put into SSA form. Memory references have been converted to uses of `access` and `update` operators as in [2]: `access(m, x)` "returns" the item at address `x` in the (version of) memory named `m`, and `update(m, x, v)` "returns" an updated version of memory `m` in which the value at address `x` is `v`. This conversion is necessary because although a *memory* can be renamed, a *memory access* as a unit cannot be: in general, there is no way to determine if two memory references `m[i]` and `m[j]` refer to the same location—in other words,

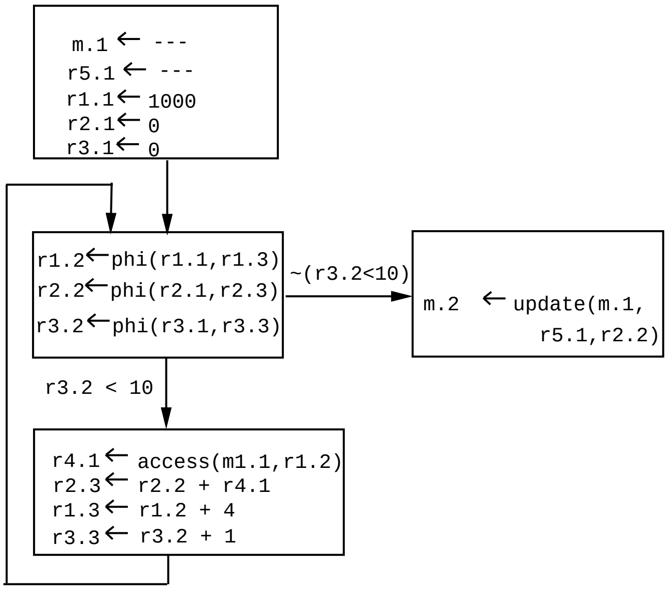

Figure 2: Flow graph for the program after SSA conversion.

to determine if `i` and `j` are equal—and so no way to consistently rename both references. Therefore, we conservatively assume that an assignment altering one memory location alters all memory locations: we always rename the `m` on the left-hand sides of `update`s. Conflicting definitions of loop registers are resolved by ϕ-assignments, which are shown as assignments with `phi(...)` on their right-hand sides in the figure. Also note the addition of *implicit assignments*, shown as assignments with `---` on their right-hand sides. This indicates that the destination was defined outside the program.

3 SSA Form and Data Dependence

A data dependence exists between two assignments if one writes to storage—either a register or memory—that the other uses. Traditional representations of programs have three kinds of data dependence [18]: *flow dependence*, *anti dependence*, and *output dependence*. If assignment A defines storage that assignment B uses, then A is a *flow supporter* of B, and B is a *flow dependent* of A. If A uses storage that B (re)defines, then A is an *anti supporter* of B, and B is an *anti dependent* of A. If A defines storage that B (re)defines, then A is an *output supporter* of B, and B is an *output dependent* of A. Computing global data dependences means determining the supporters and dependents of each assignment in the program. This information is used in most optimizations—dead assignment elimination and common subexpression elimination being two examples—as well as in combining.

SSA conversion eliminates anti and output dependences involving most registers by making each the target of just one assignment. However, it does not remove anti dependences involving memory or *pinned registers*, which are logical registers that *must* be assigned a particular physical register. (Memory and pinned registers will be collectively referred to as *pinned storage*.) For example, assuming that `SP` represents the stack pointer, the anti dependence in

```
r1 ← SP.
SP ← SP + 4.
```

continues to exist in

```
r1.1 ← SP.1.
SP.2 ← SP.1 + 4.
```

despite SSA conversion, since both `SP.1` and `SP.2` *must* be assigned the same physical register, the stack pointer. Similarly, there is an anti dependence in

```
r2.1 ← access(m.1, r1.1).
m.2 ← update(m.1, r3.1, 0).
```

involving `m.1` and `m.2`, because the second assignment might write to the same memory location that the first assignment read.

It is worth noting that anti dependences spring from our assumptions concerning storage. For example, the assumption that `m.2` and `m.1` correspond to the same physical storage implies that there is an anti dependence. However, if the `update` operator could return an updated *copy* of memory, then there would be no anti dependence, since `m.2` and `m.1` would no longer refer to the same storage; copying an array is a well-known technique for breaking anti dependences [19]. It is pinned storage—storage that cannot be reallocated—that causes anti dependences. There are no anti dependences involving unpinned storage such as normal logical registers, since the register allocator maintains the illusion of infinitely many registers. Thus, storage allocation options influence anti dependences, as mentioned briefly in [5] with regard to arrays. Since the RTL System (or any other optimizer) cannot copy all of memory each time an `update` is done, it must keep track of anti dependences. We describe the technique we have developed for doing so next.

SSA form makes keeping track of flow dependences simple, and they are the only kind of dependence that most optimizations need to know about. The RTL System calculates flow dependences when it puts a program into SSA form, and the optimizations it performs update this information. Anti dependences are inspected relatively infrequently: in the RTL System, roughly one-tenth as often as flow dependences. Thus, instead of being maintained explicitly, anti dependence information is computed from flow dependence information as needed. For a transfer t, this is done using the relationships[1]

$$antiCandidates(t) \;=\; \left(\bigcup_{x \,\in\, flowSupporters(t)} flowDependents(x) \right) - t$$

[1] Technically, these relationships do not yield all anti dependents and supporters, but only the nearest ones. This is sufficient, since satisfying anti dependences involving these automatically satisfies anti dependences involving others.

$$
\begin{aligned}
\mathit{antiSupporters}(t) &= \{x \mid x \in \mathit{antiCandidates}(t) \wedge \mathit{trashes}(x,t)\} \\
\mathit{antiDependents}(t) &= \{x \mid x \in \mathit{antiCandidates}(t) \wedge \mathit{trashes}(t,x)\}
\end{aligned}
$$

where the predicate $\mathit{trashes}(a, b)$ is true iff a writes to physical storage that b reads, regardless of the logical names involved: even though `SP.1` and `SP.2` are logically distinct in our example, they represent the same physical storage, so $\mathit{trashes}(\texttt{SP.2} \leftarrow \texttt{SP.1 + 4, r1.1} \leftarrow \texttt{SP.1})$ is true. More precisely,

$$
\mathit{trashes}(a, b) \equiv \bigvee_{w \in \mathit{write}(a),\, r \in \mathit{read}(b)} \mathit{samePhysicalStorage}(w, r)
$$

where $\mathit{read}(x)$ and $\mathit{write}(x)$ are the set of logical storage units respectively read and written by x, and $\mathit{samePhysicalStorage}()$ is true if its arguments *must* be assigned the same physical storage; it is presumably represented somewhere as a table indexed by the possible types of arguments.

These relationships hold only because we require that *an assignment use any pinned storage that it defines*: as demonstrated in the previous examples, any assignment that sets a version of `SP` or `m` also uses a version of them in order to conform to this rule. (For the rare assignment that writes `SP` without reading it, we use an `update()` operator for registers similar to the one for memory.) The set $\mathit{antiCandidates}(t)$ usually has few members, since $\mathit{flowSupporters}(t)$ nearly always has only one or two members, and a given assignment generally has only a few flow dependents. Thus, the computation of anti dependences takes little time. This technique is much simpler than the one suggested in [4] for finding anti dependences.

4 SSA Form and Peephole Optimization

The combining phase of the RTL System attempts to combine several register transfers into a single one corresponding to machine instruction, and is good at finding complex addressing modes. One common processor having such addressing modes is the Motorola 68020, which we will use in our examples.

As in PO, the combining phase scans the program once, combining each transfer with members of its *window*—flow supporters, and possibly their flow dependents—and replacing the original transfers with the result if it corresponds to a machine instruction. For example, combining the third register transfer with the second (its flow supporter) in the RTL fragment

```
r2 ← r1 + 8.
r4 ← r3 + r2.
r5 ← *r4.
```

would yield `r5 ← *(r3 + r2)`, which does not correspond to a 68020 instruction. However, combining this new transfer with the first transfer in the fragment (one of the flow supporters of the new transfer) yields `r5 ← *(r1 + r2 + 8)`, which does correspond to an instruction. The RTL System would then replace the third transfer above with this new transfer, and delete the other two.

PO also uses a window to form instructions. The difference is that in the RTL System the dependence information automatically determines the window, while PO has to explicitly manage the window, which we found difficult to implement correctly. Consider the following example, taken from [7]:

```
1        r10.1 ← access(m.1, r1.1, r2.1).
2        r11.1 ← 3.
3        r10.2 ← r10.1 + r11.1.
4        m.2 ← update(m.1, r3.1, r10.2).
5        m.3 ← update(m.2, r4.1, r10.2).
```

In PO, although transfer 5 uses `r10.2`, its window must not contain transfer 3 because of the intervening use in transfer 4: if transfer 3 were in the window, Combiner would attempt to collapse the pair, possibly improperly rearranging the code so that the effect of transfer 3 occurs after transfer 4. We found that such exceptions to the window rules complicated both window management and the combining algorithm. In the RTL System, there is a flow dependence from transfer 3 to transfer 5, but the combining algorithm is prevented from putting transfer 3 after transfer 4 because of the additional flow dependence from transfer 3 to transfer 4. Thus, although SSA form was invented for the purpose of optimization, it also makes combining simpler.

Like PO, the RTL System has an instruction recognizer that examines the structure of a register transfer to tell whether it represents a legal machine instruction. It uses a table of patterns generated from a machine description, with each pattern describing a machine instruction.[2] Patterns are parameterized by the registers and constants that are operands of instructions, so

[2] Thanks to the hashing technique it uses for instruction recognition, the RTL System's

each pattern can match many different register transfers. For example, the "add register" instruction for the 68020 is represented by the pattern `r$0` $\leftarrow$ `r$0 + r$1`. The two-address instruction format of the 68020 means that the pattern must likewise be two-address. However, SSA form requires that each register be the destination of exactly one assignment, so no register transfer could match the pattern. Thus, SSA form conflicts with any code generator based on a machine description, since it requires transfers that cannot match the patterns derived from the machine description for a two-address machine.

A register transfer like `r2` $\leftarrow$ `r1 + r0` can be implemented by a two-address add instruction if either `r1` or `r0` is dead after the register transfer. If `r1` is dead then `r1` and `r2` can be assigned the same physical register and the two-address add will implement the register transfer. If neither `r1` nor `r0` are dead then the register transfer will have to be implemented by two instructions, one of which is a copy.

We resolve the conflict between SSA form and instruction recognition by making patterns three-address but letting them impose constraints on register assignment; this is similar to a technique described in [3] for solving the same problem. Thus, the pattern for the add instruction would be `r$0` $\leftarrow$ `r$1 + r$2` with the constraint that `r$0` and `r$1` must be the same or that `r$0` and `r$2` must be the same. Register assignment must satisfy these constraints, possibly by introducing register-to-register copies. For example, assuming that the values in `r12` and `r11` are not needed afterwards, the fragment

```
r13 ← r12 + r11.
r14 ← r12 − r11.
```

would be translated to the 68020 code

```
mov.l d1,d5
add.l d2,d1
sub.l d5,d2
```

In this case, the constraint that `r13` be assigned the same physical register as either `r12` or `r11` causes a copy from `d1`, the physical register corresponding

combining phase takes only about 10% of total execution time, whereas PO's Combiner took about 50% because it used a bottom-up parsing technique. Since instruction recognition is already so fast, we have not bothered to implement a more modern instruction recognizer.

to `r11`, to be inserted, since neither `r12` nor `r11` are dead after the first statement. They are dead after the second statement, however, and so the constraint that `r14` be assigned the same physical register as either `r12` or `r11` can be satisfied by using `d2`, the physical register corresponding to `r12` at that point.

At first glance, it might seem that there is a danger of inserting many unnecessary register copies. However, this fear is unfounded, since the copies that are inserted are just the ones that would appear anyway even in two-address intermediate code: they serve to store values for later use, and so are intrinsic to the computation.

5 Conclusion

We have constructed a working code optimizer that uses SSA form. The RTL System is being used in TS, an optimizing compiler for Smalltalk [16]. An earlier version was used in Javalina, an environment for programming digital signal processors [15]. It generates code for the Motorola 68020, the National 32000, the SPARC, and several digital signal processors.

SSA form has made the RTL System simpler by eliminating multiple variable definitions, and more powerful by providing sophisticated optimization algorithms. SSA form does not directly address the question of anti dependences, and it makes using a table-driven code generator less straightforward than it would be otherwise, but the techniques described in this paper can be used to easily overcome these problems without affecting the simplicity and power of SSA form.

References

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989.

[2] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[3] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[4] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1991.

[5] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth 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.

[6] Ron Cytron, Andy Lowry, and Kenneth Zadeck. Code motion of control structures in high-level languages. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–85, 1986.

[7] Jack W. Davidson. *Simplifying Code Generation Through Peephole Optimizations*. PhD thesis, University of Arizona, 1981.

[8] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, 1984.

[9] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, April 1980.

[10] M. Ganapathi and C. N. Fischer. Integrating code generation and peephole optimization. *Acta Informatica*, 25:85–109, 1988.

[11] M. Ganapathi, C. N. Fischer, and J. L. Hennessy. Retargetable compiler code generation. *ACM Computing Surveys*, 14(4):573–592, December 1982.

[12] R. S. Glanville and S. L. Graham. A new method for compiler code generation. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 509–514, 1978.

[13] S. L. Graham. Table-driven code generation. *IEEE Computer*, 13(8):25–34, August 1980.

[14] S. L. Graham, R. R. Henry, and R. A. Schulman. An experiment in table-driven code generation. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices*, pages 32–43, 1982.

[15] Kurt J. Hebel. *An Environment for the Development of Digital Signal Processing Software*. PhD thesis, University of Illinois at Urbana-Champaign, 1989.

[16] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An optimizing compiler for Smalltalk. In *Proceedings of OOPSLA '88, Object-Oriented Programming Systems, Languages and Applications*, pages 18–26, November 1988. printed as SIGPLAN Notices, 23(11).

[17] Ralph E. Johnson and Carl McConnell. *The RTL System: A Framework for Code Optimization*. Technical Report UIUCDCS-R-9-1698, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1991.

[18] D. J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, 1978.

[19] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[20] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[21] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.