Eliminating Redundant Object Code†

Jack W. Davidson‡
Christopher W. Fraser

Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

Compilers usually eliminate common subexpressions in intermediate code, not object code. This reduces machine-dependence but misses the machine-dependent common subexpressions introduced by the last phases of code expansion. This paper describes a machine-independent procedure for eliminating machine-specific common subexpressions. It also identifies dead variables, defines windows for a companion peephole optimizer, and forms the basis of a retargetable register allocator. Its techniques for handling machine-specific data should generalize to other optimizations as well.

1. Introduction

Most implementations of common subexpression elimination ('CSE') accept intermediate code such as triples or quadruples, not object code. This simplifies implementation but may sacrifice code quality because the eventual expansion of the intermediate code may introduce new common subexpressions. For example, expanding address calculations often creates redundant machine-dependent subexpressions that cannot be eliminated in the intermediate code because they do not appear until after the final stages of code generation. This problem may be attacked by carefully tailoring the intermediate code to the machines at hand, but this complicates the design of the intermediate code and jeopardizes its machine-independence.

†This work was supported in part by the National Science Foundation under Grant MCS-7802545.
‡Present address: Jack W. Davidson, Dept. of Applied Mathematics and Computer Science, University of Virginia, Charlottesville, VA 22901

This paper describes two programs — called cacher and assigner — that implement CSE and register allocation on object code for an optimizing Y compiler [Hanson]. They treat machine-specific instructions, but they do so in a machine-independent fashion. By generalizing an existing CSE algorithm [Freiburghouse, Sites], cacher attacks the problems described above, identifies dead variables, and defines windows for a companion peephole optimizer [Davidson]. With assigner, it implements register allocation. The current implementation operates locally, but its techniques for handling machine-specific data should generalize to global optimizations as well.

2. Register Transfers

Like many recent retargetable code generators and optimizers [Cattell, Davidson, Glanville], cacher represents instructions using ISP-like register transfers [Bell]. For example, the instruction sequence

```
r[1] = r[1] + m[a]
m[b] = r[1]
pc = if cc > 0 then loop else pc
```

adds memory location a to register 1, stores the result in memory location b, and jumps to loop if register cc exceeds zero. The code generator emits these instead of equivalent assembly or machine code. Because cacher and assigner assume responsibility for register allocation, the code generator may assume an infinite supply of registers. assigner maps references to nonexistent pseudo-registers (like r[1000]) onto real registers. This is a common technique for simplifying code generation [Chaitin, Gries].

3. cacher

cacher symbolically simulates register transfers and records the values that they store. When it encounters one that recomputes an existing value, it edits the instruction stream to reuse the value stored earlier.

To cacher, a *block* is a section of code with exactly one entry. Two symbolic expressions (or 's-exprs') *a*

and *b* are said to *match* at a given point in a block if and only if the block has set variables so that *a* and *b* have the same value at that point. For example, after the sequence

```
r[1] = m[a]
r[1] = r[1] + m[b]
```

r[1] matches m[a]+m[b]. One s-expr is said to be *older* than another at a given point in a block if and only if it was computed earlier in the block. For example, after the instructions above, m[a] is older than m[a]+m[b], which is older than r[1].

cacher reads a block at a time, processes it, and emits the simplified code. As it advances through the block, it maintains a cache of the s-exprs that are computed. It partitions this cache into equivalence classes induced by the equivalence relation *match*, and it sorts the s-exprs in each equivalence class by age, oldest first. The cache associates each active s-expr with the list of s-exprs it now matches, making it easy to identify incoming s-exprs that have been computed already. The procedure follows:

1. Split each register transfer into two parts. Call its left-hand side *dst* and its right-hand side *src*.
2. Find the oldest expression that matches *src*. That is, for each register that appears in *src*, replace the register in *src* with the oldest value in the register's equivalence class. Call the resulting string *csrc*.
3. Find the oldest expression that matches *dst*'s address. First, strip off the leading name and outermost brackets, if any, exposing the address calculation. Call this string *addr*. Replace each register in *addr* with the oldest value in the register's equivalence class. Call the result *caddr*. Restore the original name and outermost brackets to *caddr*, and call the result *cdst*.
4. Find the cheapest replacement for *src* and *addr*. If *src* has been computed before then some previous instruction has had the same *csrc* and has entered it in the cache (see Step 5). Thus the cheapest expression for this *src* is the cheapest s-expr in *csrc*'s equivalence class. If a cheaper expression for *src* is found, it replaces *src* in the original register transfer. Otherwise, *src* is not disturbed. A similar procedure replaces *addr* with its cheapest equivalent. Cheapness is determined by a machine-dependent function described below.
5. Update the cache to reflect the current instruction's change to *dst*. Deletions are needed because this instruction changes *cdst* and thus invalidates s-exprs that use it. Insertions are needed because this instruction forms a new equivalence, between *csrc* and *cdst*. First, find *csrc*'s equivalence class. If no equivalence class contains *csrc*, create one. Next, delete from the cache every s-expr with which *cdst* interferes. Finally, add *cdst* to the equivalence class found for *csrc* above. *cdst*'s new home is identified before the deletions because it cannot be found later if *csrc* is among the deletions. Interference is determined by a machine-dependent function described below.

cacher also creates and maintains *use lists*, which link the instructions that use each particular s-expr. When an instruction is first encountered, it is added to the use lists of each s-expr it uses. When an instruction is changed to use a cheaper reference, it is removed from the use lists of the s-exprs it had used, and it is added to the use lists of the new s-exprs that it now uses. When a register's use list becomes empty, the instruction that loaded that register is deleted and removed from the use lists of the s-exprs *it* had used. This may trigger further deletions and removals, recursively. This prompt removal from use lists assumes that the code generator uses each temporary just once. This simplifies management of temporaries for all parties yet sacrifices nothing because the code generator may use as many registers as it needs.

Finally, cacher may be asked to keep some values *out* of registers. For example, the PDP-11 has so few allocatable registers that it may be better to save them for values harder to access than, say, constants. Code generators implement this by marking instructions that load such values. cacher treats marked instructions like all others, except that it will not reuse their *cdsts* in step 4 above. A marked instruction may, however, be deleted if it provides input to a larger redundant computation.

Though simple, cacher does several optimizations at once. Among these are redundant load elimination, common subexpression elimination, dead-variable identification, and peephole definition.

3.1 Redundant Load Elimination

Since register references are cheaper than memory references, the algorithm above will remove loads. For example,

```
a = c
b = c + 1
```

might compile into code that begins

```
r[1] = m[c]
m[a] = r[1]
r[2] = m[c]
...
```

The first two instructions cache three s-exprs — r[1], m[a], and m[c] — which form one equivalence class. As the third instruction is processed, *dst* and *cdst* become r[2], and *src* and *csrc* become m[c]. There is an equivalence class that contains m[c], and the cheapest member of this set is r[1], so the third instruction is changed to

```
r[2] = r[1]
```

eliminating a redundant load of memory location c.

This example exposes one of cacher's assumptions. In theory, a machine might have instructions for loading from memory but not for inter-register transfers like the one above. Thus, in theory, cacher should use the peephole optimizer's instruction checker [Davidson] to verify the legality of its changes. In practice, most machines allow register references to replace memory references, so instruction checking has not yet proven necessary.

3.2 Common Subexpression Elimination

Deleting unused instructions eliminates common subexpressions. For example, the code

```
a = c + 3
b = c + 3
```

might compile into

```
1. r[1] = m[c]
2. r[2] = 3
3. r[1] = r[1] + r[2]
4. m[a] = r[1]
5. r[3] = m[c]
6. r[4] = 3
7. r[3] = r[3] + r[4]
8. m[b] = r[3]
```

When cacher processes instruction 6, it recognizes that r[2] already holds 3, so it changes the instruction to use the cheaper r[2] and adds it to r[2]'s use list. It then processes instruction 7 and discovers that the sum m[c]+3 is already available in r[1]. It changes the instruction to use the cheaper reference, adds the instruction to r[1]'s use list, and removes it from the use lists of r[3] and r[4]. This empties these lists, so instructions 5 and 6 are deleted. It then turns to instruction 8, replaces r[3] with the cheaper r[1] (older s-exprs are the cheaper of equals), deletes the now-unused instruction 7, and yields the program:

```
1. r[1] = m[c]
2. r[2] = 3
3. r[1] = r[1] + r[2]
4. m[a] = r[1]
8. m[e] = r[1]
```

Common subexpressions are usually eliminated at a higher level, but machine-level CSE can do more because *all* values are exposed at this level. For example, address calculations often require code to multiply indices, shift offsets, or add frame pointers. Expanding similar address calculations may create redundant multiplications, shifts, or additions that cannot be eliminated earlier because they do not appear earlier. For example, the source program

```
f(a[i])
 ...
i = i*2
```

might generate the machine-independent postfix code

```
push  i
push  a
index
call  f
 ...
push  i
push  2
mul
pop   i
```

There are no obvious common subexpressions in this code, but expanding the index for byte-addressed machines may form the same product (or shift result) computed by the mul. It is hard for conventional code generators to catch such common subexpressions without introducing machine-dependencies into their CSE code.

Also, some instructions have side effects (e.g., divisions often yield a remainder as well as a quotient) that cannot be used in higher-level CSE because they do not appear at a higher level. Similarly, expanding comparisons requires, on some machines, a subtraction followed by a comparison with zero. The difference may be redundant, but it cannot be recognized as such at a higher level because it does not appear until after the machine-dependent stages of code expansion. The situations that create machine-specific common subexpressions are ad hoc and hard to enumerate, but they occur nonetheless.

3.3 Window Definition

cacher also defines windows for a peephole optimizer. Most peephole optimizers use a fixed window and thus consider many pairs that cannot combine and miss many valid combinations merely because the instructions are not adjacent. Scanning for more distant candidates [Wulf] may consider many instructions that are unlikely to combine.

cacher exploits the observation that many peephole optimizations combine an instruction that sets some cell with the next instruction that uses it†. cacher employs its use lists to link such instructions for its companion peephole optimizer, which tries combining only linked instructions. This improvement on the fixed window typically makes the peephole optimizer run 30% faster and yield code that is 20% shorter. Code inspection uncovers few missed peephole optimizations.

†When the cell is the program counter, it is next 'used' by the instruction after the branch *and* by the instruction targeted by the branch. This definition of program counter 'use' allows peephole optimizers to collapse branch chains and eliminate unreachable code [Davidson].

3.4 Dead-variable Identification

cacher records where cells are last used, so it passes this information to the peephole optimizer and the register assigner, which can better combine instructions and assign registers if they know where cells die. Compilers usually identify dead variables earlier, but just as machine-level CSE permits a few new optimizations, so does machine-level dead-variable analysis. For example, many calling sequences return function values in a fixed register. After the function return, the calling sequence often moves the value to another register, in case the special register is needed again. If it is not needed again, this move will prove unnecessary. Conventional compilers identify such avoidable moves deep in a machine-dependent code generator. cacher identifies such moves with a far more general operation.

4. Register Assignment

assigner maps the pseudo-registers onto the real registers. It assigns a real register to each pseudo-register, and it replaces each use of the pseudo-register with the associated real one. It frees the real register when the pseudo-register dies.

When the demand for hardware registers exceeds the supply, assigner allocates a temporary, saves the contents of the least-recently-used hardware register, and frees it for use. The LRU replacement policy is sub-optimal [Freiburghouse], and the new loads and stores could introduce inefficiencies, but these shortcomings have not yet earned attention: the 3500-line Y compiler is compiled using only 42 register spills for the PDP-11 (with three allocated registers) and none for the DECsystem-10 (with twelve).

The register assigner also translates the register transfers to assembly code. It uses a machine-independent algorithm that is driven by a machine description [Davidson].

5. Implementation

cacher and assigner are written in C [Kernighan] and run on a PDP-11/70 under UNIX. cacher is 1100 lines of code and processes about 100 instructions per second. assigner is 310 lines of code and processes about 140 instructions per second. Neither has been much optimized, so these rates can probably be improved. Because information is not maintained across labels, neither program requires much memory.

cacher and assigner make typical object programs 5-10% smaller and 10-15% faster. In addition, cacher's window definition makes the compiler's peephole optimizer run about 30% faster and yield code that is 20% shorter than the version of the peephole optimizer that uses a fixed window. The three programs work together to allow, for example, a naive code generator for the PDP-11 to yield code that is typically at least as good as that produced by UNIX's machine-dependent C compiler. Similar results have been observed for implementations for the DECsystem-10, the CDC Cyber 175, and the Intel 8080.

cacher is retargeted by replacing the patterns that identify register names and by revising the functions that determine cost and interference. The cost function accepts two s-exprs and returns the cheaper. Usually, s-exprs matching a register pattern are preferred to those matching simple memory references, which are preferred to those matching indirect memory references. This function must be revised for each machine, but the change typically effects fewer than ten lines of code.

The interference function reports a conflict when assignment to *cdst* invalidates a cache entry *src*. This happens when *cdst* appears in *src*, when *cdst* indexes an array used in *src*, when *cdst* indexes a global array and *src* uses parameter array, and when *cdst* indexes a parameter array and *src* uses global array. These rules involve fewer than ten lines of machine-dependent code, though languages with more opportunities for aliasing [Aho] than Y (e.g., pointers) might require a few more.

assigner is retargeted by replacing the patterns that identify the names and numbers of the machine registers and by giving code templates for loading and storing such registers. These changes are typically simpler than those made to cacher.

6. Discussion

Conventional code generators optimize as early as possible. This often simplifies the requisite analysis and avoids machine-dependence, but it may sacrifice some code quality. Whenever an intermediate code is expanded, it is possible for the expansion to introduce optimizable patterns that will be missed by 'early' optimizers. Experience with cacher shows that at least one optimization traditionally applied to machine-independent triples or quadruples can be applied at reasonable cost to equivalent register transfers. It is now natural to seek other optimizations that can be usefully applied to object code. For example, address expansion often produces code that can be moved out of loops and that needs global register allocation. Work in progress is adapting such existing optimizations to the machine level, but other optimizations may merit similar treatment.

Acknowledgment

Dave Hanson provided useful advice and parts of the Y compiler.

References

A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

C. G. Bell and A. Newell. *Computer Structures: Readings and Examples.* McGraw-Hill, 1971.

R. G. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems 2*(2):173-190, April 1980.

G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages 6*(1):47-57, January 1981.

J. W. Davidson and C. W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems 2*(2):191-202, April 1980.

R. A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM 17*(11):638-642, November 1974.

R. S. Glanville and S. L. Graham. A new method for compiler code generation. *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*:231-240, January 1978.

D. Gries. *Compiler Construction for Digital Computers.* Wiley, 1971.

D. R. Hanson. The Y programming language. *SIGPLAN Notices 16*(2):59-68, February 1981.

B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

R. L. Sites. Machine-independent register allocation. *SIGPLAN Notices 14*(8):221-225, August 1979.

W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke. *The Design of an Optimizing Compiler.* American Elsevier, 1975.