Code Selection through Object Code Optimization

JACK W. DAVIDSON
University of Virginia
and
CHRISTOPHER W. FRASER
University of Arizona

This paper shows how thorough object code optimization has simplified a compiler and made it easy to retarget. The code generator forgoes case analysis and emits naive code that is improved by a retargetable object code optimizer. With this technique, cross-compilers have been built for seven machines, some in as few as three person days. These cross-compilers emit code comparable to host-specific compilers.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation; compilers; optimization*

General Terms: Languages

Additional Keywords and Phrases: Code generation, compilation, optimization, peephole optimization, portability

1. INTRODUCTION

Most compilers perform lexical, syntactic, and semantic analysis, code optimization, and code generation, in this order [1]. Code optimization includes common subexpression elimination, constant folding, code motion, and strength reduction [1, 37]. Code generation includes register allocation, code selection, and perhaps some peephole optimization [1]. Most optimizers treat a high-level intermediate code to avoid machine dependencies. Most peephole optimizers are machine specific and ad hoc, so they are normally confined to correcting a handful of patterns that cannot be more easily corrected before code generation [4, 23, 24, 37].

This paper describes a compiler that optimizes *after* code generation [8]. A naive code generator emits "worst case" code that is subsequently improved by an object code optimizer. The object code optimizer is machine independent and

This work was supported in part by the National Science Foundation under Grant MCS-7802545.
Authors' addresses: J. W. Davidson, Dept. of Applied Mathematics and Computer Science, Univ. of Virginia, Charlottesville, VA 22901; C. W. Fraser, Dept. of Computer Science, Univ. of Arizona, Tucson, AZ 85721.

general. It extracts its machine specifics from a machine description, and it implements only two general optimizations: eliminating common subexpressions and replacing adjacent instructions with equivalent singletons. The optimizer allows the use of simple code generators and makes the compiler easy to retarget. Cross-compilers for seven machines have been built, some in as few as three days. They emit code comparable to that from typical machine-specific compilers.

The compiler, called "YC", translates the Y programming language [18], though its techniques apply to other Algol-like languages as well. Indeed, its optimizer and code generation strategy have been used to build compilers for subsets of Modula II [36] and a guarded command language [28]. Section 2 describes YC's organization, Section 3 describes its performance, and Section 4 compares this work with similar work.

2. ORGANIZATION

Figure 1 shows YC's five phases. Like many retargetable compilers [22, 26], YC's front end compiles source code into an abstract machine code. The next phase expands this into *register transfers* [5], a representation roughly equivalent to assembly code. The *Cacher* phase eliminates common subexpressions in register transfers, the *Combiner* phase replaces sequences of register transfer instructions with equivalent singletons, and the *Assigner* phase translates them into assembly code. Cacher, Combiner, and Assigner extend an earlier optimizer called *PO* [7].

YC uses abstract machine code to keep the front end machine independent, and it uses register transfers as a machine-independent representation for machine-specific instructions. Register transfers describe the effect of machine instructions. They have the form of conventional expressions and assignments over the hardware's storage cells.[1] For example, if the instruction represented by the assembly code

```
load x
```

loads the accumulator with the memory cell labeled x, then it might be represented with the register transfer

```
ac = m[x]
```

Any particular register transfer is machine specific, but the *form* of register transfers is machine independent. YC uses register transfers because their machine-independent form permits it to optimize machine-specific code in a machine-independent way. For example, a machine-independent algorithm can identify even machine-specific common subexpressions in register transfers. Such machine-independent optimizations take the place of classical machine-specific case analysis and thus improve retargetability.

Given a machine description like that in Section 2.4, it is simple to perform syntax-directed translation between register transfers and assembly code [7, 10]. Thus YC could substitute assembly code for register transfers in interphase communication. However, register transfers are not appreciably harder to emit than assembly code, so this extra step is omitted in the interest of efficiency.

[1] Thus YC's register transfers are not suited to describe instructions so complex as to have internal loops (like block moves). Such instructions require separate treatment [25].

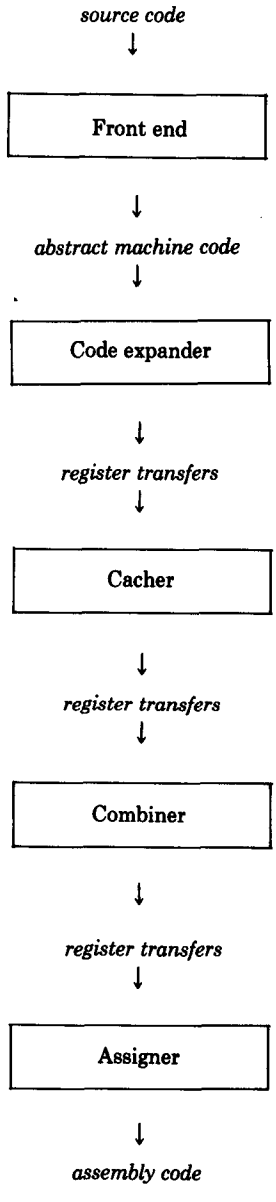

Figure 1

The five subsections below elaborate on YC's five phases. Other documents focus more closely on Cacher [9] and on early versions of Combiner [7, 10]. The following subsections summarize the operation of these phases and show how they collaborate to simplify code generation.

2.1 Front End

Like many compilers, YC's scanner and parser are completely machine independent, and the only machine dependencies in its semantic analyzer are three arguments that define the relative widths of the basic integer, character, and

floating-point data types. Like many portable compilers [22, 26], YC's semantic analyzer emits abstract machine code. However, thorough object code optimization makes it practical to use an abstract machine that is particularly simple.

Abstract machines are usually implemented through a process logically equivalent to macro expansion. These code expanders may track register contents to avoid redundant loads, and they may try to avoid a few inefficient juxtapositions in the object code, but the final code expansion process is typically quite local. This encourages abstract machine instructions that are "large" enough to collect features from many intended target machines. Consider the following examples:

(1) Some machines have *special-case* instructions like clear instructions that are cheaper than stores, increment and decrement instructions that are cheaper than adds and subtracts, and tests against zero that are cheaper than arbitrary comparisons. If some intended target machine has such special-case instructions, then it is common to include similar instructions in the abstract machine, so that the code generators can generate the target instructions without checking operands.
(2) If some intended target machine has three-address adds, then the abstract machine may be given three-address adds, so that the code generators need not examine more than one abstract machine instruction to select the proper add.
(3) If some intended target machine has an *autoincrement* (which dereferences a pointer and automatically advances it to the next item), the abstract machine may be given a similar or "larger" feature (e.g., built-in vector operations). The code generators would otherwise need to examine two or more abstract machine instructions to use autoincrements.

Such abstract machines might be termed *union machines* because they offer a set of features roughly equivalent to the union of the sets of features offered by typical machines. This definition is elaborated below.

YC achieves local code quality through optimization instead of case checking, so its abstract machine need not be a union machine. The abstract machine uses a simple postfix code, has no registers, and supports about the smallest set of operators, types, addressing modes, and structures required by the source language. For example, the translation of

```
a[i] = b[i]
```

is little more than a translation into postfix:

```
pushga i    load i's address
pushi       load i's value
pushga a    load a's base address
index 2     form a[i]'s address, given that elements have width 2
            (i.e., add 2*i to a's base address)
pushga i    repeat this process for b[i] ...
pushi       :
pushga b    :
index 2
pushi       load b[i]'s value
popi        store it into a[i]
```

YC's abstract machine has only one significant union machine feature. Most of its abstract machine instructions that treat addresses have two variants, one for conventional addresses and another for addresses of elements of a character array. This is necessary to allow packed character arrays on word-addressed machines, where conventional addresses are not enough to address a packed element. Thus YC's abstract machine does not have union machine operations, but it does have a union machine data type. It needs one because its optimizer can improve code but not storage formats. Appendix A summarizes the abstract machine's instructions.

In contrast with union machines, an abstract machine like YC's might be termed an *intersection machine* because it offers a set of features roughly equivalent to the intersection of the sets of features offered by typical machines. That is, an intersection machine is that subset of machine functions that can be expected of all target machines.[2] This makes intersection machines similar to reduced-instruction-set computers (*RISCS*) [29, 30], which have been shown to simplify compilers. Intersection machines need not use postfix, though, like RISCs, they should use a small, regular instruction set.

Absolute minimality is not necessary in intersection machines. For example, an abstract machine could force the front end to compile all conditionals into combinations of branch-if-zero and branch-if-positive instructions. However, this would complicate the front end without significantly simplifying the implementation of the abstract machine. Thus a practical intersection machine includes all the usual arithmetic, logical, and conditional operations, but it includes only one form of each and allows the front end to emit simple code without case analysis.

Indeed, absolute minimality is not even useful. Technically, any effective procedure can be computed by a machine with only one instruction (e.g., to subtract two cells and branch if the result is positive). Thus the ultimate intersection machine has just one instruction, and, conversely, the ultimate union machine interprets source code directly. All practical abstract machines are on a continuum between these extremes. Just as practical RISCs are "impure" for they can always be further reduced, so few practical union or intersection machines are "pure." They are defined as the union and intersection of *typical* machines, not abstract contrivances.

Intersection machines are easy to implement. For example, YC's abstract machine has 99 opcodes, whereas P-code [27], a union machine for Pascal compilers, has 219 opcodes and tends to emit more object code per abstract machine instruction. Wulf notes that, to simplify compilers, computer architectures should offer "precisely one way to do something, or all ways should be possible" [38]. Intersection machines normally have exactly one obvious implementation for each source language construct, so they effectively meet this requirement. Simplifying the abstract machine simplifies semantic analyzers that emit its code and code expanders that implement it. This helps make a compiler reliable, easy to understand, and easy to modify.

[2] This definition is due to an anonymous referee.

Intersection machines are stable. For example, few union machines exploit the VAX-11's new queue-manipulation operations. Queuing extensions might be added, but the need for such changes makes union machines unstable. In contrast, intersection machines are less sensitive to target machine changes. YC's abstract machine may not include queuing instructions, but YC's optimizer can introduce them into the object code where appropriate. YC's abstract machine and YC's first two implementations (targeting the PDP-11 and the DECSystem-10) were developed in parallel. Since this initial phase, YC's abstract machine design has been stable over five additional machines. Minor changes have been made to accommodate languages other than Y, but no changes have been made to accommodate new machines.

Unfortunately, intersection machines require verbose code sequences. Verbose abstract machine code usually yields verbose target machine code, which no doubt explains the historical dominance of union machines. Theoretically, any simple compiler based on an intersection machine would be retargetable, but its code would be unacceptable without object code optimization, and classical object code optimizers are hard to retarget. Sections 2.3 and 2.4 show how YC's object code optimizer overcomes this obstacle.

These points recall the attempt to define a universal intermediate language or "UNCOL" [32, 33], which was to have solved the "M*N" compiler problem. Given M languages and N machines, M*N compilers are required if each directly compiles source code into machine code. With UNCOL, only M + N compilers were to have been required: M to compile source code to UNCOL, plus N to compile UNCOL to machine code. Unfortunately, no truly universal UNCOL has emerged to handle all languages and machines. This is partly because UNCOLs traditionally had to be union machines to achieve reasonable performance. Optimizing object code removes this constraint and thus extends the range of any particular UNCOL. Such optimization helps an UNCOL accommodate a wide range of machines, but it is not yet clear whether it helps accommodate a wide range of languages. In particular, compiling very-high-level languages into very-low-level code may discard efficiency clues that present optimizers cannot recover. YC shows that is not the case for Algol-like languages, but the question remains open for other languages.

2.2 Code Expander

Most code generators perform much case analysis. They must supply at least one code pattern for each abstract machine instruction, and, for "large" abstract machine instructions, they often need different patterns for different classes of operands.

Special-case instructions add complexity. The hardware may not have a special-case instruction for some abstract machine special-case instruction. Here, the semantic analyzer's case analysis was wasted (on this machine), and an otherwise unnecessary pattern must be written to use a more general hardware instruction. Conversely, the hardware may have special cases that were not anticipated by the designer of the abstract machine. Here, case analysis is necessary to detect those instances of, say, the abstract machine's general compare instruction that can use a particular "skip-if-zero" hardware instruction.

To yield good code, most code generators also store and check contextual data. For example, the DECSystem-10 has register-to-register, register-to-memory, and memory-to-register adds. The right choice for any particular addition depends on which, if either, of its operands is available in a register, and which, if any, of its operands or result should remain in a register. Such machine-specific context checking works against a clean separation of instruction selection from register allocation and assignment, and forms a substantial component of the typical hand-written code generator.

YC leaves all such case analysis to the object code optimizer. Each language operation has only one image on the abstract machine, and each abstract machine operation has only one image on each target machine. YC also leaves register allocation to the object code optimizer. Its code generators assume an infinity of pseudoregisters, which the object code optimizer maps onto the available supply.

This makes YC's code expanders about as simple as is possible. The code expanders take only two actions to simplify subsequent optimization. First, the code expanders emit register transfers instead of assembly code, though this adds little complexity. Second, the code expanders flag cells that are obviously dead. For example, the code expander knows that any condition code registers are used only after comparisons, so it marks such registers as dead as it generates code for other operations. This allows, for example, subsequent phases to implement some additions with an autoincrement that does not set the condition code instead of a more costly add instruction that does.

The program below gives the PDP-11 code for the example from Section 2.1. It shows how naive the code selection can be.

Input	Output	Comment
pushga i	r[12] = i	Load i's address.
pushi	r[13] = m[r[12]]	Load i's value
pushga a	r[14] = a	Load a's address.
index 2	r[15] = r[13]	Copy i,
	r[15] = r[15] ≪ 1	and shift to form byte offset.
	r[16] = r[15]	Copy offset,
	r[16] = r[16] + r[14]	and form a[i]'s address.
pushga i	r[17] = i	Repeat for b[i]
pushi	r[20] = m[r[17]]	⋮
pushga b	r[21] = b	⋮
index 2	r[22] = r[20]	
	r[22] = r[22] ≪ 1	
	r[23] = r[22]	
	r[23] = r[23] + r[21]	
pushi	r[24] = m[r[23]]	Load b[i]'s value,
popi	m[r[16]] = r[24]	and store it into a[i]

Each value is formed in a new register so that each value will be available during common subexpression elimination. For example, a[i]'s address is formed in r[16] to preserve the shifted value of i in r[15]. Common subexpression elimination will reuse r[15] in the formation of b[i]'s address, and computations used only once will be absorbed into larger instructions wherever possible. The code above is verbose. Cacher and Combiner collaborate to make it usable.

Each of YC's target machines needs its own code expander, but the lack of redundancy and case analysis makes code expanders easy to implement. Each is

a large case statement with one case per abstract machine instruction, and each case is logically equivalent to a macro. Theoretically, the code expander must be completely rewritten for each new machine. In practice, a new code expander is written by editing one for another machine. Since the code expanders generate similar patterns on different machines (mainly register-to-register operations and simple addressing), and since register transfers can be more similar than assembly codes (e.g., "r[x]" is often used to name registers), their realization is similar on different machines. Thus many of the edits affect not logic but the skeletal register transfers that addresses and register indices instantiate. The seven code expanders written to date are each about 650 lines of C.

2.3 Cacher

Cacher eliminates common subexpressions in blocks of register transfers bounded by labels in the input program. It uses a conventional algorithm [11], which effectively simulates the register transfers and records the value taken on by each register. When it finds a register taking on an already available value, it reuses the previously computed value and deletes the instructions that compute the new one. For example, this removes four instructions from the PDP-11 program above:

Input	Output	Comment
r[12] = i	r[12] = i	Load i's address.
r[13] = m[r[12]]	r[13] = m[r[12]]	Load i's value.
r[14] = a	r[14] = a	Load a's address.
r[15] = r[13]	r[15] = r[13]	Copy i,
r[15] = r[15] ≪ 1	r[15] = r[15] ≪ 1	and shift to form byte offset.
r[16] = r[15]	r[16] = r[15]	Copy offset,
r[16] = r[16] + r[14]	r[16] = r[16] + r[14]	and form a[i]'s address.
r[17] = i		Repeat for b[i]
r[20] = m[r[17]]		⋮
r[21] = b	r[21] = b	
r[22] = r[20]		
r[22] = r[22] ≪ 1		
r[23] = r[22]	r[23] = r[15]	⋯ but reuse byte offset ⋯
r[23] = r[23] + r[21]	r[23] = r[23] + r[21]	
r[24] = m[r[23]]	r[24] = m[r[23]]	Load b[i]'s value,
m[r[16]] = r[24]	m[r[16]] = r[24]	and store it into a[i]

Cacher also flags each block's last use of each cell so that Combiner will know when it has deleted such a use and can then delete the instruction that sets the now-unused cell, and so that Assigner will know when to release a register assigned to a cell. An example making use of these data appears below.

Cacher also links each instruction to the first instruction that uses one of its results. Benchmarks show that Combiner typically runs 30 percent faster and yields code that is 20 percent shorter when it tries combining only such "logically" adjacent instructions rather than the more obvious physical adjacencies.

Cacher is retargeted by changing three routines, which, respectively, identify the names of registers to track, extract such register names from larger strings, and compare the efficiency of two expressions. For most machines, this last function merely prefers expressions consisting of a single register reference to all other expressions.

Common subexpression elimination is not much harder to perform on register transfers than on a more regular structure like quadruples, and it is possible to catch common subexpressions after code generation that cannot be caught before because they are machine specific. Expanding abstract machine code for address calculations often creates such machine-specific common subexpressions.

2.4 Combiner

Combiner advances over Cacher's register transfers seeking adjacent instructions that can be replaced with singletons. It symbolically simulates adjacent pairs and triples[3] of instructions to learn their combined effect and searches a machine description for an instruction with this combined effect. If it finds one, it replaces the original instructions with the singleton.

Symbolic simulation is a string operation. To simplify the searching of machine descriptions, instructions that make several register transfers are represented as though the transfers were done in parallel, not sequentially. Thus Combiner computes the combined effect of two instructions by substituting the values assigned to cells in the first for appearances of those cells in the second. Consider, for example, the first two PDP-11 instructions above:

```
r[12] = i
r[13] = m[r[12]]
```

Combiner computes their combined effect by replacing the r[12] in the second instruction with the value assigned to r[12] in the first, and then concatenating the two effects. This yields

```
r[12] = i; r[13] = m[i]
```

This algorithm treats assignments to the program counter as a special case. If the first instruction branches, the register transfers from the second are made conditional on the branch not being taken. If the first instruction branches unconditionally, this effectively removes unreachable code. Combiner also simulates such branches with their targets, which often collapses branch chains.

Next, Combiner removes from this combined effect any register transfers that set dead variables. In the example above, Cacher will have flagged this spot as the last use of r[12], so Combiner can simplify the initial combined effect to

```
r[13] = m[i]
```

At this point, Combiner also performs a few symbolic simplifications. For example, if symbolic simulation creates an expression like r[1] + 0, symbolic simplification deletes the "+0" to simplify instruction recognition. A few of Combiner's simplifications involve labels (e.g., those that eliminate a branch to the label on the next statement) and thus may need changing for an assembler with an unusual label syntax. The other simplications are machine independent. Occasionally a new machine requires a machine-independent simplification to

[3] Combiner simulates triples because many machines with singletons equivalent to a load/operate/store sequence do not have singletons for the load/operate and operate/store subsequences. Thus Combiner needs a three-instruction window to catch these optimizations, though no pressing need for a larger fixed window has emerged.

correct a pattern that does not occur on other machines. For example, one simplification replaces POP(PUSH(x)) with x. The need for such simplifications is machine specific, but the simplifications themselves are machine independent.

Combiner presents the simplified register transfer to a finite automaton that recognizes the legal instructions. This automaton is automatically constructed from a formal machine description, described below. If no match is found, Combiner advances to the next instruction and tries combining it with its logical predecessors. If a match is found—and, in the example above, one is, though the dead-variable elimination was necessary—Combiner replaces the original instructions with the combined effect and tries combining the new instruction with its logical predecessors. Combiner reaches back instead of forward to find adjacencies so as to cascade optimizations in one pass without backup.

Optimizations routinely cascade. Consider this (logical, not physical) subsequence of the sample program above:

```
r[14] = a
        :
r[16] = r[15]
r[16] = r[16] + r[14]
        :
m[r[16]] = r[24]
```

Combiner reduces it to one instruction in three steps:

After step 1	After step 2	After step 3
r[16] = r[15]	r[16] = r[15]	
r[16] = r[16] + a	:	
:		
m[r[16]] = r[24]	m[r[16] + a] = r[24]	m[r[15] + a] = r[24]

Eventually, the original 16 instructions are reduced to the optimal three:

```
r[15] = m[i]
r[15] = r[15] ≪ 1
m[r[15] + a] = m[r[15] + b]
```

This example shows how repeated object code optimizations can substitute for careful case selection. The "case analysis" has been done by an automatically generated finite automaton.

This example is not unusual; indeed, one 17-to-1 reduction has been reported. It is possible to build a machine with instructions for the naive sequences and the singletons but without the intermediate-level instructions Combiner needs as it reduces a six-instruction sequence to five instructions, then four, then three, etc. Giegerich proposes an improvement on Combiner that eliminates its narrow window and accommodates such machines [14]. Fortunately, such machines seldom appear in practice. Designers that include "large" and "small" instructions seem to include "medium" ones as well, presumably because it is easy to do so once the "big" instructions have been included.[4]

[4] Only once—for the above-mentioned VAX queue instructions—has a crucial "medium" instruction

Classical peephole optimizers correct only those sequences that match a few hand-written, machine-specific patterns. In contrast, Combiner has no ad hoc case analysis: it combines *all* possible adjacent pairs and triples. As a result, its effect can be described formally and concisely: when it is finished, no one-, two-, or three-instruction sequence can be replaced with a cheaper single instruction having the same effect.

In the presence of naive code generation, Combiner's machine description and patternless substitution algorithm appear preferable to a classical peephole optimizer. For example, when compiling YC's front end for the PDP-11, Combiner has been observed to make about 300 distinct types of replacements, each involving two or three input instructions and one output instruction. A collection of 300 patterns to perform the same replacements would be more complex, less complete, and harder to check [35] than the 100-line PDP-11 machine description. Also, the number of possible patterns grows rapidly with machine complexity. If A is the number of addressing modes and I the number of instructions, then a pattern list of only two-input patterns may grow as $(A*I)^2$ where, as shown below, a machine description grows as A + I.

Combiner is retargeted by supplying a new machine description. The machine description is processed into tables that are compiled with Combiner. These tables isolate Combiner's machine-specific data. Combiner's code is machine independent just as a table-driven parser is language independent. Since Combiner is machine independent, it is not tuned to improve code quality on a new machine. Machine descriptions are occasionally tuned (see the discussion of trade-offs below), but Combiner is not.

Each machine description is a grammar for syntax-directed translation between assembly language and register transfers. For example, the production

```
NZ = DSTW ? 0; :=: tst DSTW
```

describes the PDP-11 "tst" instruction, which compares a word with zero and sets the condition code register ("NZ") accordingly. The register transfer appears on the left and the equivalent assembly code on the right. Similarly, the production

```
DSTW = DSTW + SRCW; NZ = DSTW + SRCW ? 0; :=: add SRCW, DSTW
```

describes the PDP-11 instruction that adds two words and sets the condition code register to describe the result. Code generators need not know the details of encoding of the condition code, so the semantics of the comparison operator ("?") need not be described. The code expander and machine description need only use the operator consistently. This technique is routinely used to suppress architectural details irrelevant to code generation.

Productions also describe the legal operands. For example, the production

```
DSTW := r[REGNO] :=: rREGNO
```

---

been missing. This problem is solved by including the missing instruction as a macroinstruction in the machine description. This solution is ad hoc, but the problem is VAX specific, so it seems better to add an instruction to the VAX-11 machine description than to the abstract machine, which must be implemented once for each target. The proposed windowless optimizers will solve this problem more elegantly.

describes register operands. They are expressed in register transfers with a string like "r[1]", and in assembly code with a string like "r1". Similarly, the production

DSTW := m[r[REGNO] + IDENT] :=: IDENT(rREGNO)

describes indexed operands. They are expressed in register transfers with a string like "m[r[1] + a]" and in assembly code with a string like "a(r1)".

Machine descriptions are occasionally tuned to achieve specific effects. For example, Combiner does not currently make time–space trade-offs. That is, it assumes that one instruction is always better than two. This assumption is usually valid but fails occasionally. There are several ways a machine description can help solve this problem. First, if some forms of some singleton are always worse than the equivalent pair or triple, then the machine description should describe only the desirable forms. Alternately, the machine description can describe a pseudoinstruction realized by an assembler macro that implements the preferred multiinstruction sequence. If the pseudoinstruction appears in the machine description before the less desirable singleton, then it will be used whenever it applies. Such ad hoc measures have sufficed so far, but ultimately it may prove necessary to include instruction costs (in space and time) in machine descriptions. Such data should be optional so that machine describers are not obliged to provide them when the usual "one instruction is better than two" heuristic holds.

2.5 Assigner

Once all optimization is complete, Assigner maps the now-reduced set of pseudoregisters onto the available hardware registers, introducing spills where needed. It also translates the register transfers to assembly code, using a finite automaton generated from the machine description. For example, this phase accepts the three register transfers above and produces the assembly code:

```
mov i,r2
asl r2
mov b(r2),a(r2)
```

Appendix B traces the compilation of a longer example.

When a register transfer is realized by more than one instruction, Assigner's automaton prefers the one listed first in the machine description. Thus machine descriptions should list faster special-case intructions first. Only Assigner uses this ordering. Combiner cares only that *some* instruction realizes a combined effect. Assigner decides *which* instruction is best.

Assigner is retargeted by supplying a function to print a label definition in the appropriate assembler syntax and by revising the register assignment module. For general register machines, the register assignment module is retargeted by changing a few tables that describe the number of registers, their names, and whether they may be allocated. Machines with unique register structures may require some recoding, but even the unique register set of the CDC Cyber was accommodated by the first author in 2–3 days. Assigner also uses Cacher's function that extracts register names from a string and Combiner's machine description.

Table I

	Y	C	Pascal
8q	37.0 seconds 306 bytes	37.3 seconds 304 bytes	62.2 seconds 492 bytes
ctoi	19.5 seconds 192 bytes	21.5 seconds 188 bytes	26.2 seconds 424 bytes
ictoi	39.5 seconds 236 bytes	45.9 seconds 244 bytes	69.6 seconds 426 bytes

3. EVALUATION

3.1 Machine Retargetability

YC runs on the VAX-11 and the PDP-11 under UNIX[5] and produces code for the VAX-11, PDP-11, DECSystem-10, CDC Cyber, Motorola 68000, IBM 370, and Intel 8080. The system has been in production use for over two years, and it is stable enough that the 68000 and 370 versions were brought up by students in a second-term compiler course. It has been distributed and is running at several sites.

YC is easily retargeted. Once the initial DECSystem-10 and PDP-11 versions had shaken out most of the bugs, the Cyber, 8080, and VAX-11 versions were brought up in no more than five person-days each, and students new to the system, the target machine, and even UNIX have retargeted YC in about 20 person-days. This includes writing a machine description, coding a naive code generator, making the simple changes to Cacher and Assigner, and assembling a skeletal runtime system for Y.

The tables below quantify the performance of the retargets on three typical programs. The program "8q" solves the eight queens problem recursively, "ctoi" converts a packed digit string into an integer, and "ictoi" converts an unpacked digit string into an integer, to expose differences in the handling of characters and integers. The tables show the size of the emitted code and its execution time. To minimize timing errors, the eight queens program is called 100 times, and the other functions are called 30,000 times. Where compilers offer different levels of optimization, the comparisons below use the level that produces the best code. No trade-offs were necessary: optimization levels that produced the shortest code also produced the fastest. All debugging options like subscript checking were turned off.

Table I compares the PDP-11 retarget with the host-specific UNIX C compiler [31], and the Vrije University portable Pascal compiler [34]. The portable UNIX C compiler [21] produces PDP-11 code similar to the host-specific C compiler's, and the published code fragments from recent description-driven code generators [12, 15] suggest that they do too. YC's code runs fastest mainly because it eliminates redundant expressions, though other factors effect the Pascal com-

[5] UNIX is a registered trademark of AT&T Bell Laboratories.

Table II

	Y (retargetable)	Y (host-specific)	Ratfor
8q	26.7 seconds 85 words	26.1 seconds 74 words	n/a n/a
ctoi	34.9 seconds 71 words	51.2 seconds 61 words	n/a n/a
ictoi	25.7 seconds 72 words	24.8 seconds 70 words	19.2 seconds 66 words

Table III

	Y	Pascal	Ratfor
8q	17.6 seconds 71 words	16.6 seconds 76 words	n/a n/a
ctoi	18.1 seconds 56 words	26.3 seconds 88 words	n/a n/a
ictoi	15.5 seconds 60 words	14.0 seconds 77 words	8.7 seconds 47 words

piler's code. Some of these factors are Pascal related and beyond the control of the compiler. For example, the Pascal compiler may use two instructions to load a character because the one instruction used by the other compilers extends the sign bit. However, at least one factor illustrates a trade-off made differently by the Y and Pascal compilers. In particular, there are patterns in the code produced by the Pascal compiler that need peephole optimization. The compiler performs peephole optimization, but it does so early, on abstract machine code. This simplifies retargeting but misses machine-specific inefficiencies introduced during code expansion.

Table II compares the DECSystem-10 retarget with a host-specific Y compiler and with a Ratfor translator that uses the host-specific DEC Fortran compiler. The Ratfor code does best mainly because the Fortran compiler performs global (interblock) register allocation. Ratfor does not offer recursion or packed character strings, so 8q and ctoi benchmarks are unavailable for this compiler. YC yields faster code for ctoi than the host-specific Y compiler because of a different code generation strategy for handling characters. Otherwise, the host-specific Y compiler does better, due to more global register allocation [19].

Table III compares the CDC Cyber retarget with the host-specific ETH Pascal compiler [2] and with a Ratfor translator that uses the host-specific CDC Fortran compiler. The CDC Fortran compiler is an ambitious optimizing compiler, though much of the speed indicated above is due to a nonrecursive calling sequence unavailable to the Y and Pascal compilers. The Pascal and Y compilers are more nearly comparable, but even they generate code quite differently. YC eliminates a few more common subexpressions, where the Pascal compiler performs more sophisticated Cyber-specific optimizations. For example, YC uses ordinary short-

Table IV

	Y	C
8q	40.4 seconds 286 bytes	41.6 seconds 307 bytes
ctoi	38.2 seconds 164 bytes	40.3 seconds 161 bytes
ictoi	40.5 seconds 198 bytes	41.1 seconds 175 bytes

circuit evaluation for relationals where the Pascal compiler uses more Boolean instructions, perhaps to avoid jumps that invalidate the instruction buffer. Also, it implements some multiples with shifts and adds, which is beyond Combiner and would have to be implemented in YC by extra case analysis in the code expander or machine description. As reflected in Table III, some of these optimizations make a program longer but faster. Some of these optimizations were tailored to the CDC 6400 and are less appropriate on the newer Cybers, though the effect of this anomaly on the timings given is small.

The Cyber retarget was the first after YC stabilized. Despite an irregular register set that complicated register allocation, this retarget took the first author five days, which included developing a skeletal runtime system. Thus the tables above show that a compiler based on thorough retargetable object code optimization can compete with a carefully tailored host-specific compiler like the ETH Pascal compiler.

Table IV compares the VAX-11 compiler with the UNIX C compiler for the VAX-11. The code produced by the two compilers differs in two notable ways. First, YC eliminates redundant expressions and the C compiler does not. Second, C loops use add-compare-and-branch instructions that cannot be exploited by Y loops because YC's front end puts the loop test at the top of the loop and the increment at the bottom. Here a seemingly machine-independent high-level code generation decision influences the generated code. The VAX-11 retarget was completed in one person-week.

YC has also been retargeted for the Intel 8080, Motorola 68000, and IBM 370 though lack of hardware has left these versions largely untested. The 8080 retarget was done by the first author in three days. The 68000 and 370 retargets were done by students in a second term compiler class with no previous experience with Y, its compiler, or, in some cases, UNIX. They took about 20 person-days each.

In summary, YC is competitive with host-specific compilers. In some cases, its code is as good as or better than its competitors. In the remaining cases, no important differences are due to code selection. Rather, they are due to calling conventions outside the control of the compiler, or to global optimizations that could be added to YC.

YC would benefit from some global optimization. The front end folds constants, Cacher eliminates common subexpressions within blocks, and Combiner replaces adjacent pairs and triples with equivalent singletons, but no other optimizations

are performed. In particular, YC omits strength reduction, execution ordering, loop optimizations (e.g., moving loop-invariant code, eliminating induction variables), and optimizations requiring global data-flow analysis (e.g., global register allocation, code hoisting) [1]. Many of these optimizations are machine independent and could be done before code expansion. Alternately, they could be performed on register transfers, which are not so different from the quadruples to which these optimizations are often applied. Because the form of register transfers is machine independent, the optimizations can be implemented machine independently. They would, however, automatically optimize machine-specific code. For example, they would automatically move *all* loop-invariant code, including machine-specific code introduced during code expansion, where optimizing quadruples moves only the machine-independent quads exposed prior to code expansion. This recommends register transfers as an intermediate code.

3.2 Language Retargetability

YC's back end has been also used to develop retargetable compilers for a subset of Modula II [36] and for a guarded command language [28]. These compilers have the same organization as YC. The front end for the guarded command language emits an intermediate code tailored to the source language, but the Modula II compiler uses YC's abstract machine with minor changes. The Modula II compiler targets the PDP-11 and uses the optimizer developed for the PDP-11 YC retarget. The other compiler targets the VAX-11 and uses the optimizer developed for the VAX-11 YC retarget. Each of these compilers was developed in a semester by a student carrying other courses.

Thus YC's back end can be used with other languages. Y has no pointers, structures, or multidimensioned arrays, but YC's abstract machine does not assume any such restrictions. Indeed, it cannot, for Y's call-by-reference requires abstract machine pointers, and Y's procedure frames are essentially structures. The only significant additions likely to be required by other Algol-like languages should be a few extended-precision types and operators for them.

Further, YC's back end can be used with other front ends. It assumes only that the front end folds constant expressions. Combiner will automatically fold constant expressions itself, but its finite automaton cannot process arbitrary context-free expressions correctly, so this should done by the front end. A more sophisticated instruction recognizer would avoid this requirement. YC's back end requires no other particular assistance from the front end, and it allows the front end to forgo case analysis.

3.3 Speed

YC's speed is tolerable but could be better. On a VAX-11/780, it processes about 10 source lines per second, where the UNIX C compiler processes 40. Some of this difference is due to extra optimizations performed by YC. Table V shows the rough fraction of time taken by each phase. YC eliminates redundant expressions, and the C compiler does not. If Cacher, YC's redundant expression eliminator, were eliminated for comparison purposes, YC would process about 15 source lines per second, to the C compiler's 40. The code expanders would have to compute the back links and dead variables, but this is easily done as the code is emitted.

Table V

Phase	Time (%)
Front end	10
Code expander	3
Cacher	31
Combiner	45
Assigner	11

The division of the compiler into five separate programs also artificially slows YC. The division was to simplify development and to accommodate larger programs in the limited PDP-11 address space. With development largely complete, it is appropriate to consider combining the phases. If the front end eliminated branch chains as it emitted code, Combiner could buffer only basic blocks and use much less storage. Profiles suggest that combining the phases would reduce compile times by 10–20 percent.

Improvements are also possible within some phases. Combiner has been tuned but not exhaustively, Cacher has been tuned only cursorily, and the other phases have not been tuned at all. Thus there is room for improvement within the current implementation. The authors are also working to make Combiner generate patterns at compile-compile time for a faster compile-time peephole optimizer. (Others have made similar proposals [14].) Experiments suggest that this optimizer may run 5–10 times faster than Combiner. This, plus conventional tuning and eliminating Cacher for comparison purposes, should give YC about the same speed as the UNIX C compiler.

4. RELATED WORK

YC is not the first compiler to emit naive code and then improve it, but its particular mix of retargetability and code quality appears to be unique. The IBM ECS/GPO [20] and PL.8 [3] compilers use a similar strategy, though YC's use of machine descriptions appears to make it faster to retarget than these compilers.

The Vrije University Pascal compiler [34] uses peephole optimization at a higher, machine-independent level to improve naive abstract machine code. Peephole optimization is easier at this level, but it misses the opportunities for optimization introduced when its (union machine) abstract machine code is later expanded. As such, it trades compiler speed for code quality: on a PDP-11/70, it processes about 10 lines/second to YC's 6, but its eight queens program takes 622 ms to YC's 370.

Finally, YC's portability suggests using its optimizer with, or as an alternative to, the new pattern-matching code generators [6, 13, 16, 17]. Each of these systems works differently, but most match intermediate code with instruction patterns and pick the one that maximizes some measure. For example, one picks the instruction that matches the most intermediate code [16, 17].

These systems generate good local code, but their code requires some peephole optimization. Indeed, one system accepts peephole optimization rules in an attribute grammar that describes the instruction set [13], though these rules are machine specific and require hand-retargeting. An optimizer like Combiner seems an appropriate companion to these systems because it is more thorough and

more easily retargeted than typical hand-coded peephole optimizers. If Combiner is to be used anyway, it will automatically allow the code generator to emit naive code and thus require only a small set of instruction patterns.

This set of patterns could be as small as the set used in YC's naive code generators. It is only slightly harder to plug these instructions into YC's naive code generators than it is to describe them for a pattern matching code generator, so a pattern-matching code generator might make YC more elegant, but it is not expected to make it much faster to retarget.

With a little extra effort, a larger set of instruction patterns could be coded for a companion pattern-matching code generator, midway between the complete instruction set and the naive subset. This would divide the responsibility for case analysis between the code generator and optimizer, reducing the sometimes large size of the code generation tables [17] and the sometimes long execution time of Combiner. Eventually, it should be possible to eliminate the need for duplicate machine descriptions by retargeting a pattern-matching code generator and object code optimizer from one machine description.

APPENDIX A. YC's Abstract Machine

The executable instructions of YC's abstract machine are summarized below, with variants indicated in parentheses. Being postfix, most instructions take no explicit operands.

add, subtract, multiply, divide, negate, convert (integer and real versions)
and, or, complement, modulus, shift right, shift left (integer only)
form address of array element
augment address
load, store a value indirectly (integer, real, character)
load constant (integer, real, address)
load address of a variable (global, local, parameter, reference parameter)
jump (simple, indexed)[6]
compare and jump conditionally (integer, real)
compare with two bounds and jump conditionally[6]
compare and generate a truth value (integer, real)[6]
establish actual argument (integer, real, character, address)
call, return (integer, real, no value)
procedure entry, exit

A few nonexecutable opcodes define labels, initialize storage cells, etc.

[6] These instructions contain a few redundant features that remain in YC's abstract machine for historical reasons. The two forms of the jump instruction could be replaced with one if the jump instruction were changed to take its target address from the stack instead of taking it as an explicit argument; the "compare with two bounds" instruction could be realized with two "compare and jump" instructions; and the "compare and generate a truth value" instructions could be realized with "compare and jump" instructions, or vice versa. These changes should simplify YC without reducing code quality, and they should yield an abstract machine without redundancies.

APPENDIX B. A Compilation Trace

This appendix traces the compilation of the program below for VAX-11.

```
# ctoi—convert string "in" to integer
integer ctoi(in)
  character in[ ]
  integer i, n

  n = 0
  i = 1
  while (in[i] >= '0' & in[i] <= '9'){
    n = 10*n + in[i] − '0'
    i = i + 1
    }
  return (n)
end
```

The first tabulation below shows the naive register transfers before Cacher, and the second traces them through the optimizations.

Before Cacher	Comment
r[21] = r[15] + n	Load n's address.
r[22] = 0	Load zero.
m[r[21]] = r[22]	Clear n.
r[23] = r[15] + i	Load i's address.
r[24] = 1	Load one.
m[r[23]] = r[24]	Set i to one.
L1: r[25] = r[15] + i	Load i's address.
r[26] = m[r[25]]	Load i's value.
r[27] = m[r[14] + in]	Load in's address.
r[30] = r[26] + r[27]	Compute in[i + 1]'s address.
r[31] = r[30] + −1	Compute in[i]'s address.
r[32] = b[r[31]]	Load in[i]'s value.
r[33] = 48	Load '0'.
NZ = r[32] ? r[33]	Compare in[i] with '0'.
PC = NZ < 0 → L3 | PC	Exit loop if less.
r[34] = r[15] + i	Load i's address.
r[35] = m[r[34]]	Load i's value.
r[36] = m[r[14] + in]	Load in's address.
r[37] = r[35] + r[36]	Compute in[i + 1]'s address.
r[40] = r[37] + −1	Compute in[i]'s address.
r[41] = b[r[40]]	Load in[i]'s value.
r[42] = 57	Load '9'.
NZ = r[41] ? r[42]	Compare in[i] with '9'.
PC = NZ > 0 → L3 | PC	Exit loop if greater.
r[43] = r[15] + n	Load n's address.
r[44] = r[15] + n	Load n's address.
r[45] = m[r[44]]	Load n's value.
r[46] = 10	Load 10.
r[47] = r[45]*r[46]	Compute 10 * n.
r[50] = r[15] + i	Load i's address.
r[51] = m[r[50]]	Load i's value.
r[52] = m[r[14] + in]	Load in's address.
r[53] = r[51] + r[52]	Compute in[i + 1]'s address.
r[54] = r[53] + −1	Compute in[i]'s address.

```
r[55] = b[r[54]]          Load in[i]'s value.
r[56] = r[47] + r[55]     Compute 10*n + in[i].
r[57] = 48                Load '0'.
r[60] = r[56] - r[57]     Compute 10*n + in[i] - '0'.
m[r[43]] = r[60]          Set n to 10*n + in[i] - '0'.
r[61] = r[15] + i         Load i's address.
r[62] = r[15] + i         Load i's address.
r[63] = m[r[62]]          Load i's value.
r[64] = 1                 Load 1.
r[65] = r[63] + r[64]     Compute i + 1.
m[r[61]] = r[65]          Set i to i + 1.
PC = L1                   Jump back to top of loop.
L3: r[66] = r[15] + n     Load n's address.
r[67] = m[r[66]]          Load n's value.
r[0] = r[67]              Set r[0] to n.
ret                       Return.
```

Before Cacher	After Cacher	After Combiner	After Assigner
r[21] = r[15] + n	r[21] = r[15] + n		
r[22] = 0	r[22] = 0		
m[r[21]] = r[22]	m[r[21]] = r[22]	m[r[15] + n] = 0	clrl n(r13)
r[23] = r[15] + i	r[23] = r[15] + i		
r[24] = 1	r[24] = 1		
m[r[23]] = r[24]	m[r[23]] = r[24]	m[r[15] + i] = 1	movl $1, i(r13)
L1: r[25] = r[15] + i	L1: r[25] = r[15] + i		
r[26] = m[r[25]]	r[26] = m[r[25]]	L1: r[26] = m[r[15] + i]	L1: movl i(r13),r2
r[27] = m[r[14] + in]	r[27] = m[r[14] + in]		
r[30] = r[26] + r[27]	r[30] = r[26] + r[27]	r[30] = r[26] + m[r[14] + in]	addl3 in(r12),r2,r3
r[31] = r[30] + −1	r[31] = r[30] + −1		
r[32] = b[r[31]]	r[32] = b[r[31]]	r[32] = b[r[30] + −1]	cvtbl −1(r3),r4
r[33] = 48	r[33] = 48		
NZ = r[32] ? r[33]	NZ = r[32] ? r[33]	NZ = r[32] ? 48	cmpl r4,$48
PC = NZ < 0 → L3 | PC	PC = NZ < 0 → L3 | PC	PC = NZ < 0 → L3 | PC	jlss L3
r[34] = r[15] + i			
r[35] = m[r[34]]			
r[36] = m[r[14] + in]			
r[37] = r[35] + r[36]			
r[40] = r[37] + −1			
r[41] = b[r[40]]	r[41] = r[32]		
r[42] = 57	r[42] = 57		
NZ = r[41] ? r[42]	NZ = r[41] ? r[42]	NZ = r[32] ? 57	cmpl r4, $57
PC = NZ > 0 → L3 | PC	PC = NZ > 0 → L3 | PC	PC = NZ > 0 → L3 | PC	jgtr L3
r[43] = r[15] + n	r[43] = r[15] + n		
r[44] = r[15] + n	r[44] = r[15] + n		
r[45] = m[r[44]]	r[45] = m[r[44]]		
r[46] = 10	r[46] = 10		
r[47] = r[45]*r[46]	r[47] = r[45]*r[46]	r[47] = m[r[15] + n]*10	mull3 $10,n(r13),r3
r[50] = r[15] + i			
r[51] = m[r[50]]			
r[52] = m[r[14] + in]			
r[53] = r[51] + r[52]			
r[54] = r[53] + −1			
r[55] = b[r[54]]	r[55] = r[32]		
r[56] = r[47] + r[55]	r[56] = r[47] + r[55]		
r[57] = 48	r[57] = 48		
r[60] = r[56] − r[57]	r[60] = r[56] − r[57]		
m[r[43]] = r[60]	m[r[43]] = r[60]	m[r[15] + n] = r[47] + r[32] − 48	movab −48(r4)[r3],n(r13)
r[61] = r[15] + i	r[61] = r[15] + i		
r[62] = r[15] + i			
r[63] = m[r[62]]	r[63] = r[26]		
r[64] = 1	r[64] = 1		
r[65] = r[63] + r[64]	r[65] = r[63] + r[64]		
m[r[61]] = r[65]	m[r[61]] = r[65]	m[r[15] + i] = r[26] + 1	moval 1(r2),i(r13)[7]

[7] This instruction would combine with the one at L1 to yield an increment instruction if there were no intervening use of register 2.

```
PC = L1                 PC = L1                 PC = L1                 jbr L1
L3: r[66] = r[15] + n   L3: r[66] = r[15] + n
r[67] = m[r[66]]        r[67] = m[r[66]]
r[0] = r[67]            r[0] = r[67]            L3: r[0] = m[r[15] + n] L3: movl n(r13),r0
ret                     ret                     ret                     ret
```

ACKNOWLEDGMENTS

Dave Hanson wrote much of YC's front end and helped with the design of the abstract machine. He and the referees offered many helpful comments on this presentation.

REFERENCES

1. AHO, A.V., AND ULLMAN, J.D. *Principles of Compiler Design.* Addison Wesley, Reading, Mass., 1977.
2. AMMANN, U. On code generation in a PASCAL compiler. *Softw. Pract. & Exper. 7*, 3 (June 1977), 391–423.
3. AUSLANDER, M., AND HOPKINS, M. An overview of the PL.8 compiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (Boston, Mass., June 23–25). ACM, New York, 1982, pp. 22–31.
4. BAGWELL, JR., J.T. Local optimizations. *SIGPLAN Not. 5*, 7 (July 1970), 52–66.
5. BELL, C.G., AND NEWELL, A. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, 1971.
6. CATTELL, R.G.G. Automatic derivation of code generators from machine descriptions. *ACM Trans. Prog. Lang. Syst. 2*, 2 (Apr. 1980), 173–190.
7. DAVIDSON, J.W., AND FRASER, C.W. The design and application of a retargetable peephole optimizer. *ACM Trans. Prog. Lang. Syst. 2*, 2 (Apr. 1980), 191–202.
8. DAVIDSON, J.W. Simplifying code generation through peephole optimization. Ph.D. dissertation, Dept. of Computer Science, Univ. of Arizona, Tucson 1981.
9. DAVIDSON, J.W., AND FRASER, C.W. Eliminating redundant object code. In *Conference Record of the 9th ACM Symposium on Principles of Programming Languages* (Albuquerque, N.M., Jan. 25–27). ACM, New York, 1982, pp. 128–132.
10. FRASER, C.W. A compact, machine-independent peephole optimizer. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages* (San Antonio, Tex., Jan. 29–31). ACM, New York, 1979, pp. 1–6.
11. FRIEBURGHOUSE, R.A. Register allocation via usage counts. *Commun. ACM 17*, 11 (Nov. 1974), 638–647.
12. GANAPATHI, M. Retargetable code generation and optimization using attribute grammars. Ph.D. dissertation, Computer Science Dept., Univ. of Wisconsin, Madison, 1980.
13. GANAPATHI, M., AND FISCHER, C.N. Description-driven code generation using attribute grammars. In *Conference Record of the 9th ACM Symposium on Principles of Programming Languages* (Albuquerque, N.M., Jan. 25–27). ACM, New York, 1982, pp. 108–119.
14. GIEGERICH, R. A formal framework for the derivation of machine-specific optimizers. *ACM Trans. Prog. Lang. Syst. 5*, 3 (July 1983), 478–498.
15. GLANVILLE, R.S. A machine independent algorithm for code generation and its use in retargetable compilers. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Sciences, Univ. of California, Berkeley, 1977.
16. GLANVILLE, R.S., AND GRAHAM, S.L. A new method for compiler code generation. In *Conference Record of the 5th ACM Symposium on Principles of Programming Languages* (Tucson, Ariz., Jan. 23–25). ACM, New York, 1978, pp. 231–240.
17. GRAHAM, S.L., HENRY, R.R., AND SCHULMAN, R.A. An experiment in table driven code generation. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (Boston, Mass., June 23–25). ACM, New York, 1982, pp. 32–43.
18. HANSON, D.R. The Y programming language. *SIGPLAN Not. 16*, 2 (Feb. 1981), 59–68.
19. HANSON, D.R. Simple code optimizations. *Softw. Pract. Exper. 13*, 18 (1983), 745–763.
20. HARRISON, W. A new strategy for code generation—The general purpose optimizing compiler. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (Los Angeles, Calif., Jan. 17–19). ACM, New York, 1977, pp. 29–37.

21. JOHNSON, S.C. A portable compiler: Theory and practice. In *Conference Record of the 5th ACM Symposium on Principles of Programming Languages* (Tucson, Ariz., Jan. 23–25). ACM, New York, 1978, pp. 97–104.
22. KORNERUP, P., KRISTEN, B.B., AND MADSEN, O.L. Interpretation and code generation based on intermediate languages. *Softw. Pract. Exper. 10*, 8 (Aug. 1980), 635–658.
23. LAMB, D.A. Construction of a peephole optimizer. *Softw. Pract. Exper. 11*, 6 (1981), 638–647.
24. MCKEEMAN, W.M. Peephole optimization. *Commun. ACM 8*, 7 (July 1965), 443–444.
25. MORGAN, T.M., AND ROWE, L.A. Analyzing exotic instructions for a retargetable code generator. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (Boston, Mass., June 23–25). ACM, New York, 1982, pp. 197–204.
26. NEWEY, M.C., POOLE, P.C., AND WAITE, W.M. Abstract machine modelling to produce portable software—A review and evaluation. *Softw. Pract. Exper. 2*, 12 (1972), 107–136.
27. NORI, K.V., AMMANN, U., JENSEN, K., NAGELI, H.H., AND JACOBI, C. Pascal-P Implementation Notes. In *Pascal—The Language and its Implementation*, D.W. Barron Ed. Wiley, 1981, 83–123.
28. PARNAS, D.L. A generalized control structure and its formal definition. *Commun. ACM 26*, 8 (Aug. 1983), 572–581.
29. PATTERSON, D.A., AND SEQUIN, C.H. A VLSI RISC, *IEEE Comput. 15*, 19 (Sept. 1982), 8–21.
30. RADIN, G. The 801 minicomputer. In Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, *SIGPLAN Not. 17*, 4 (Apr. 1982), 39–47.
31. RITCHIE, D.M. A Tour through the UNIX C Compiler. In *UNIX Programmer's Manual*, Vol. IIB, Bell Laboratories, January 1979.
32. STEEL, T.B. A first version of UNCOL, In *Western Joint Computer Conference Proceedings*, May 1961, 371–378.
33. STRONG, J., WEGSTEIN, J., TRITTER, A., OLSZTYN, J., MOCK, O., AND STEEL, T. The problem of programming communication with changing machines: A proposed solution. *Commun. ACM 1*, 8 (Aug. 1958), 12–18.
34. TANENBAUM, A.S., VAN STAVEREN, H., AND STEVENSON, J.W. Using peephole optimization on intermediate code, *ACM Trans. Prog. Lang. Syst. 4*, 1 (Jan. 1982), 21–36.
35. WAITE, W.M., AND GOOS, G. *Compiler Construction.* Springer-Verlag, New York, 1984.
36. WIRTH, N. Modula: A language for modular programming. *Softw. Pract. Exper. 7*, 1 (1977), 3–35.
37. WULF, W., JOHNSSON, R.K., WEINSTOCK, C.B., HOBBS, S.O., AND GESCHKE, C.M. *The Design of an Optimizing Compiler.* Elsevier-North Holland, New York 1975.
38. WULF, W.A. Compilers and computer architecture. *IEEE Comput. 14*, 7 July 1981, 41–47.

Received November 1982; revised September 1983; accepted January 1984