Four Generations of Portable C Compiler

David M. Kristol

AT&T Information Systems
Summit, NJ 07901

ABSTRACT

In nearly ten years' time the original Portable C Compiler (**PCC**) has evolved through three further generations that have improved its speed, generality, and code quality. Each generation uses a different code generation strategy. These compiler technologies, named **PCC2**, **QCC**, and **RCC** are all available in current AT&T C compilation system products, notably those for the AT&T 3B2 computer, which is based on the WE™ 32100 microprocessor.

The C language recognizer, or *front end*, part of **PCC2**, **QCC**, and **RCC** is identical. (The code is common among them.) It is essentially the same front end as that used in **PCC** except that it has better error checking and greater robustness. The principal differences, then, are in the respective code generation strategies, and they are the topic of this paper.

INTRODUCTION

The Portable C Compiler has been the base for uncounted numbers of C compilers on a wide variety of machines in its nearly ten years' existence. The reasons for its success are simple. Steve Johnson, author of both **PCC** and **PCC2**, put it this way: "The compiler is efficient enough, and produces good enough code, to serve as a production compiler."[1] (The C compilers that were part of UNIX* System V, Release 2, were **PCC**-based.) What's more, **PCC**'s portability means that "if you need a C compiler written for a machine with a reasonable architecture, the compiler is already three quarters finished."[1]

Compiler development within AT&T did not cease with **PCC.** We have continued to investigate and to use new compiler technology. After reviewing some of the internal details of **PCC**, this paper will describe some of the newly developed technology that has found its way into compilers for, among other machines, the WE™ 32100 microprocessor, the heart of the AT&T 3B2 computer.

The improvements that I will describe concern the strategies that the compiler uses to generate code. For the rest of this paper I will assume that the parser portion of the compiler has produced a tree-structured representation of C expressions, and that the code generator must produce the assembly language that will realize the expression's semantic intention. The new technologies, **PCC2**, **QCC**, and **RCC**, share a C language parser that is closely related to **PCC**'s, except for improved correctness and error detection.

THE PROBLEM

Code generation is easy when a machine has an infinite number of general registers, each of which may hold any datum. Of course, there are no real machines that meet this ideal, or that even come close. (That's one reason there is still a market for compiler writers.) Architecture designs have a finite, usually small, number of machine registers, and sometimes certain functions are tied to specific registers. Because machine registers must be considered scarce resources, how they are allocated, and in what order they are used, has a significant effect on

---

* UNIX is a trademark of AT&T.

the quality of code that the compiler generates. The PCC Family history is a collection of solutions to the register allocation and code ordering problem.

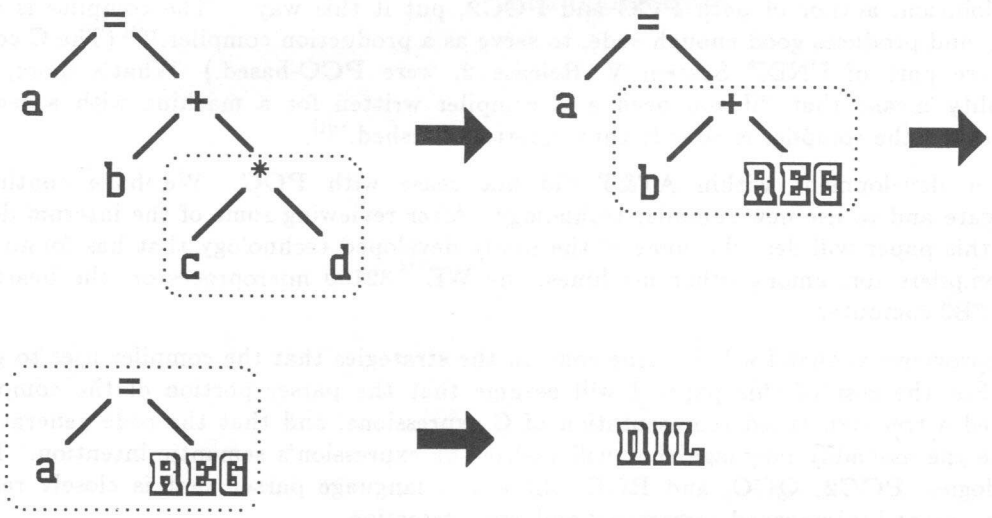
PCC FAMILY ARCHITECTURE

The PCC Family compilers have two main pieces: a (relatively) machine-independent front-end, or parser, that builds what are called *C-trees*; and a table-driven interpretive back-end, or code generator that is machine-*dependent*. To create a new compiler instance, an implementor provides definitions for symbols that describe the sizes and alignments of various data types, the byte ordering and the stack layout of the target machine, and a set of *templates* that describes what code the compiler must generate for various C operators. The templates contain the C operator and *shapes* that describe each of the operator's operands. Shapes are most easily thought of as the C-trees that correspond to an architecture's address modes.

The PCC Family back-ends consume the C-trees by starting at the fringe of the tree and working back to the root. Example 1 shows how this process works on the simple expression

```
a = b + c * d;
```

(REG represents a register node in the tree.) Part of the tree's fringe is matched by a template, the corresponding assembly code is emitted, and the portion of the tree that was matched is rewritten to be the template's result, if any, usually a scratch register. Thus the code generation and tree rewriting mirrors the execution-time behavior of the code that is generated.



Example 1. Reduction of a C expression.

PCC —— THE PORTABLE C COMPILER

Steve Johnson wrote the Portable C Compiler, **PCC**, in 1977 as part of an experiment to port UNIX from a PDP-11† base to an Interdata 8/32 computer.[2] It became possibly the most widely

used compiler that can generate native machine code. **PCC** established the compiler design framework that continues in the PCC Family.

PCC Code Generation

PCC's code generation is based on theoretical work by Sethi and Ullman[3] and others. This work describes how code generation may be ordered to use the smallest possible number of machine registers. In [1], Johnson describes how **PCC** partitions the code generation problem into determining the proper order and then following it.

The compiler implementor supplies a function to calculate the so-called Sethi-Ullman (or SU) numbers. The machine-independent part of the back-end calls this routine to populate an expression tree with SU numbers that indicate how many registers would be needed to do that part of the tree. The number at the root of the tree must be no greater than the number of scratch machine registers. Otherwise there are too few registers, and the compiler must rewrite (*spill*) part of the tree so its result will be in a temporary location, rather than in a machine register. Once the compiler successfully populates the tree with SU numbers such that no number is greater than the number of scratch registers, code generation proceeds. The code generator walks down the tree, doing higher SU numbers before lower, until it reaches leaves, at which point it begins emitting code.

This code generation approach works well most of the time. However, the Sethi-Ullman computation routine tends to look like black magic, since when it assigns numbers of registers for computations it must anticipate what templates will be used at any point and what tree rewrites will occur. When templates get added, removed, or just moved, the SU computation often has to be modified to correspond.

Rarely could the SU computation be completely accurate, so it had to strike a careful balance between optimism and pessimism. On one hand, if the SU computation is too pessimistic, the generated code will be low quality, with many spills. On the other hand, if it is too optimistic, the back-end can discover that fewer registers are available than the SU number promised, and it will quit with a compiler error.

Another weak spot in **PCC**'s design was the handling of shapes. When **PCC** was designed, "indirect through the sum of a register and offset" was considered an exotic address mode. This address mode, which came to be known as an *OREG*, is a good description for such things as stack locations and structure references, where the pointer to the structure is in a register. Each time it emits code, **PCC** must reexamine the tree to recognize if an OREG has resulted from the rewrite. When fancier architectures emerged, like the VAX and Motorola 68000[1], that had double indexing, **PCC** compilers had to scramble to fit such indexing into the OREG mold.

To summarize, **PCC** was the first practical, portable compiler technology that became widely available. It can generate good quality code, although making it do so is challenging for the implementor.

PCC2 —— CHILD OF PCC

Aho and Johnson did further work on code generation theory[4], and in 1978 Johnson produced a revised **PCC**, which became known as **PCC2**. The major changes from **PCC** were these:

- Code generation tree matching occurs top-down.

---

† PDP and VAX are trademarks of Digital Equipment Corporation.

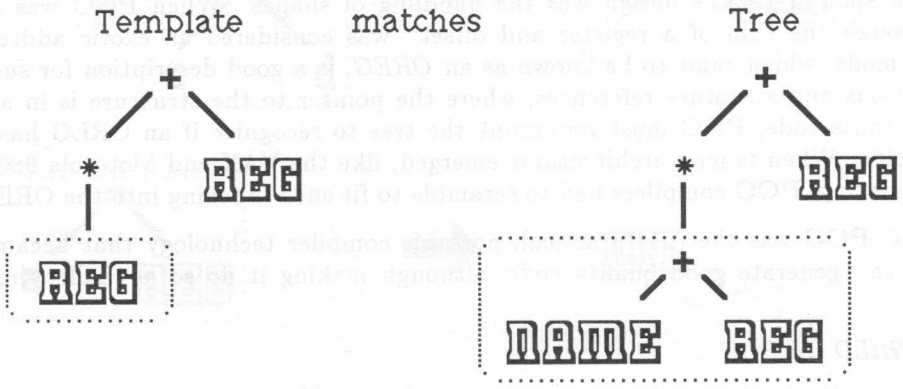
1. The AT&T UNIX PC is based on a Motorola 68010.

- Implementor-specified costs control code generation.
- Templates may contain arbitrary, implementor-specified shapes.
- The implementor uses a descriptive file to specify templates, shapes, and costs.

The Aho and Johnson theory states that the cost of a computation may be partitioned into the cost of an operation and the cost of obtaining the operands. This may apply recursively, because obtaining the operands may entail further computation. To minimize the cost of a computation, one need only minimize the sum of the cost of the operands and the operation at each step.

The **PCC2** back-end walks the expression tree, and, using the implementor supplied costs, it identifies the least expensive way to generate code for it, using a dynamic programming algorithm. Since the cost calculation algorithm considers the templates that would actually be used at each step, the calculated costs are accurate. Furthermore, the cost calculations take into account the number of registers available at each step, so register allocation is a by-product. Register spilling is another by-product because its cost is considered as a possibility at each step. This approach solves the problem of the separation between SU computation and code generation from which **PCC** suffered. However, the dynamic programming theory only allowed for one type of register, a shortcoming that was later addressed by **RCC**.

PCC2 introduces the idea of treating register shapes as *wildcards*. During its top-down tree match, **PCC2** attempts to match a template against the expression tree. If it finds that a register in a shape of the template matches a non-register in the tree, **PCC2** considers the result to be a (wildcard) match, with the following proviso: it must now calculate the *sub-tree* whose root is the mismatched non-register and assume that the result of that computation will end up in a register. At that point the register node in the shape will indeed match a register, and code generation for the original template may proceed. Example 2 depicts the case of a register (REG) node in a template matching part of a tree as a wildcard. The NAME node represents an external variable.



Example 2. Register matches as wildcard.

The theory behind **PCC2** assumes that all registers are equivalent and can contain any datum. Indeed, using registers in shapes as wildcards reinforces that need. However, by carefully designing templates and the supporting machine-dependent code, implementors have been able

to create **PCC2** compilers for machines that do not conform to this restriction, such as the Motorola 68000 and Intel 8086[2].

The machine description file that Johnson introduced for **PCC2**, commonly called *stin* (*s*hape and *t*emplate *in*put), is much more concise and easy to maintain than the previous **PCC** template representation, an initialization of a large array of structures that had to be maintained by hand. *Stin* descriptions are processed by a program to produce the **PCC2** equivalent of the **PCC** array of structures.

PCC2 compilers produce locally excellent code, but they do so slowly because of the cost calculations, which take two passes through each expression tree. In the first pass the back-end determines the least-cost code sequence. In the second pass, the back-end follows that sequence and generates code accordingly.

PCC2's *stin* file led to better compiler maintainability than **PCC** afforded. Within my group at AT&T we wanted to replace all of our **PCC**-based compilers with **PCC2**-based ones. However, because **PCC2** was as much as three times slower, we could not justify such an impact on our customers. We therefore sought to improve **PCC2**'s speed and still retain its other advantages. The result was **QCC**.

QCC —— A FASTER PCC2

QCC is based on a principle that arose from a simple observation: most of the time **PCC2** explores a great many possible code generation sequences before it chooses the obvious one. In other words, most of the time the cost-based algorithm is unnecessary. Rob Murray and I therefore replaced the cost-based algorithm with one that made the obvious choice, when there was one. The only time that a choice was "un-obvious" was when the algorithm needed more scratch registers than it had available, in which case it chose a simple spill-to-temporary algorithm and tried again.

The algorithm is a top-down tree match like **PCC2**'s. Unlike **PCC2** however, **QCC** uses the first template (*i.e.*, operation) for which the operand shapes (*i.e.*, address modes) match. We also chose the first matching shape. A register node in a shape, if it doesn't actually match a register in the C-tree, is treated as a *wildcard*, as in **PCC2**. **QCC** knows that before it can use a template that has wildcard registers in a shape, it must generate the instructions that will actually load those registers. At that point it descends the tree to generate that code first.

Two important observations must be made here. First, since we choose the *first* matching template, the order of those templates is important. In **PCC2**, you will recall, the compiler chose the lowest cost alternative from all alternatives, so the order of templates was irrelevant.

The second observation regards shapes. It would seem beneficial for **QCC** to match as large a piece of a tree as possible with a shape. Implicit in that assumption is that a machine's hardware can do the operations (notably address addition) better (and cheaper) than a sequence of equivalent instructions. This matching of larger shapes in preference to smaller ones has been called "maximal munch," because the compiler tries to bite off as large a piece of the expression tree as possible.

To get the most from **QCC**'s first-match algorithm, then, the compiler must examine templates and shapes in a preferred order. Because the description of templates and shapes was already embodied in the *stin* file, and that file was already processed by a program (called *sty*), it was natural to enhance *sty* to order the shapes and templates. In fact, one original design goal for **QCC** was to be able to take a **PCC2** compiler and convert it to a **QCC** compiler with a few

2. The Intel 8086 is the processor in the AT&T PC 6300.

quick strokes.

That goal was not entirely met, and the reasons were both good and bad. On the good side, we could do some things in **QCC** that were impossible in **PCC2**, and that reduced the number of templates an implementor had to write and simplified others. On the bad side, *sty* was not 100% successful at ordering templates, and a little hand tuning often improved code quality.

On balance, the **QCC** experiment was a tremendous success. **QCC** compilers actually turned out to be faster than equivalent **PCC** compilers. The quality of generated code fell between that of **PCC** and **PCC2**, but very close to the higher quality **PCC2** side. It still suffered from one shortcoming of **PCC2**: it could handle only one kind of machine register.

RCC —— QCC WITH REGISTER SETS

The restriction of **QCC** to one kind of machine register threatened to hamper its usefulness. As **QCC** was emerging, so was the WE 32106 Math Acceleration Unit (MAU) floating point co-processor for the WE 32100 microprocessor. Like many such co-processors, the MAU has its own set of registers, and they are emphatically *not* general purpose. They are for floating point calculations. Although we felt that we could use various tricks in **QCC** to generate code for the 32100/32106 combination, we preferred a more direct approach. This motivation led to my development of **RCC**.

RCC is **QCC** with more register bookkeeping. Where **QCC** keeps track of *how many* registers it has available to it as it descends a C-tree to generate code, **RCC** keeps track of *which ones*, too. In other words, **RCC** does a heuristic register allocation "on the fly," during tree matching.

An **RCC** implementor must divide a machine's registers into two groups (as with **PCC**, **PCC2**, and **QCC**): *scratch* and *user*. *User* registers are the ones that are available to the C programmer as `register` variables. The compiler manages and allocates the scratch registers for intermediate computations. Beyond the user/scratch division, an implementor is free to partition the registers in any useful manner. For example, on the 32100/32106 combination, three CPU and one MAU register are designated as scratch, and six CPU and two MAU registers are designated as user registers.

RCC also extends the notion of *wildcard* a bit further. In **RCC**, a register node in a tree might match a register node in a shape as a *wildcard* if the register in the tree is not one of the ones expected by the shape. For example, an implementor for an Intel 80286[3] compiler can write a template in which the shape corresponding to the shift count is precisely register `CX`. **RCC** guarantees that if that template gets used, `CX` will, in fact, contain the shift count. If the result of a shift count computation ends up in `AX`, **RCC** will see to it that the value gets moved to `CX`. Moreover, **RCC** will actually attempt to have the computation leave its result in `CX` in the first place.

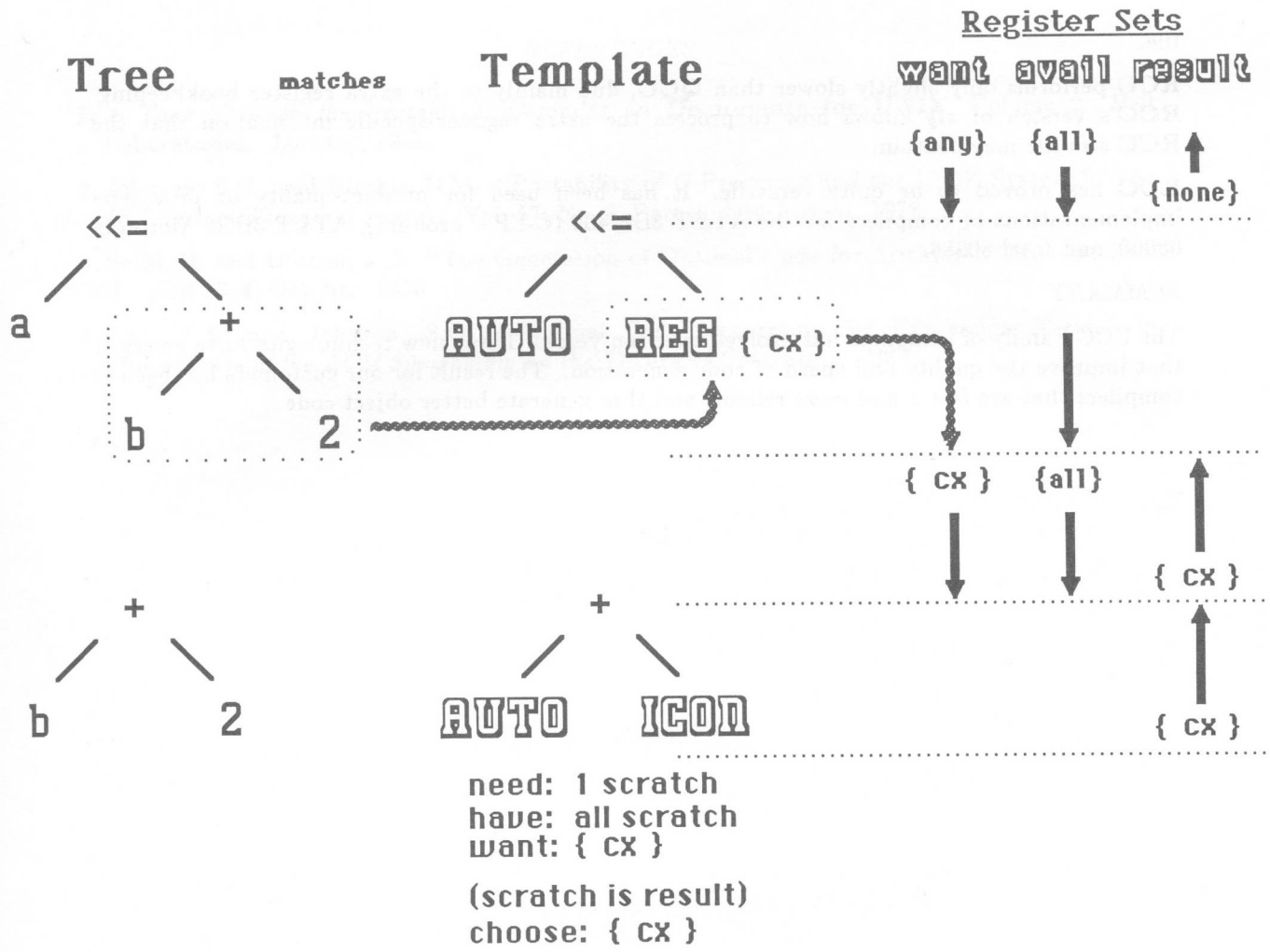
Example 3 shows how **RCC** generates code for the expression

```
a <<= b + 2;
```

for the Intel 80286, where `a` and `b` are `automatic` variables. As I stated above, the shift count must be in register `CX`, and the first template shows that the right operand indeed must be `CX`.

The rest of the example shows the register accounting that takes place. *Want* and *available* sets of registers are passed down the tree during code generation, and a *result* set of registers gets

3. The AT&T PC 6300+ uses the Intel 80286 processor.



Example 3. RCC Register Accounting

passed upward. Thus, when the back-end goes to generate code for the + operations, it knows that CX is the preferred result register. Since the template must allocate one register for its computation; since that register becomes the result of the addition; and since CX is available, **RCC** chooses CX as the scratch (and result) register for the +. At the next higher level, the pre-condition for the <<= template, loading CX, has been met, and code for the shift may be generated.

RCC lets the compiler implementor specify numbers and types of scratch registers with the same precision as registers in shapes may be specified. For example suppose a scratch register is needed for some calculation on the 32100/32106 combination. Should it be a general purpose (32100) register or a MAU (32106) floating point register? **RCC** can't guess the right answer, but the implementor can say explicitly which one to use with an appropriate notation in the *stin*

file.

RCC performs only slightly slower than **QCC**, due mainly to the extra register bookkeeping. **RCC**'s version of *sty* knows how to process the extra register-specific information that the **RCC** *stin* file may contain.

RCC has proved to be quite versatile. It has been used for product-quality or prototype implementations of compilers for the AT&T 3B2/400 (C-FP+ product), AT&T 3B20, Motorola 68000, and Intel 80286.

SUMMARY

The PCC Family of compilers has evolved over ten years. Three new technologies have emerged that improve the quality and speed of code generation. The result for our customers has been C compilers that are faster and more reliable and that generate better object code.

REFERENCES

1. *A Tour Through the Portable C Compiler*, in **Documents for UNIX**, Volume 2. Bell Laboratories. January, 1981.

2. Johnson, S.C., and Ritchie, D.M. "Portability of C Programs and the UNIX System." *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, July-August, 1978.

3. Sethi, R. and Ullman, J.D. "The Generation of Optimal Code for Arithmetic Expressions." *J. ACM* **17**,**4**, October, 1970.

4. Aho, A.V. and Johnson, S.C. "Optimal Code Generation for Expression Trees," in *Proceedings of the ACM Symposium on the Theory of Computing*. 1975.