

The Design of Very Fast Portable Compilers

Andrew S. Tanenbaum

M. Frans Kaashoek

Koen G. Langendoen*

Ceriel J.H. Jacobs

Department of Mathematics and Computer Science

Vrije Universiteit

Amsterdam, The Netherlands

Email: ast@cs.vu.nl

ABSTRACT

The Amsterdam Compiler Kit is a widely used compiler building system. Up until now, the emphasis has been on producing good object code. In this paper we describe recent work that has focused on reducing compile time. The techniques described in this paper have resulted in C compilers for the Sun-3 and VAX that are 3 to 4 times faster than the native compilers provided by the manufacturers.

1. INTRODUCTION

The Amsterdam Compiler Kit (ACK) (Tanenbaum et al., 1983) is a tool kit for building compilers. Very briefly, a source program is run through a program called a *front end*, which converts it to an intermediate language called *EM*. This language resembles the assembly language for an abstract stack machine.

The EM version of the program can then be run through a peephole and a global optimizer to improve code quality. With or without optimization, the EM code is then fed into a program called a *back end* that converts the EM code to the assembly language of the target machine. This too, can be optimized, and then assembled to the final binary executable program. The back end, final optimizer, and assembler are machine and language-independent programs that are driven by machine-specific tables. The various steps resemble a UNIX[†] pipeline, as shown in Fig. 1.

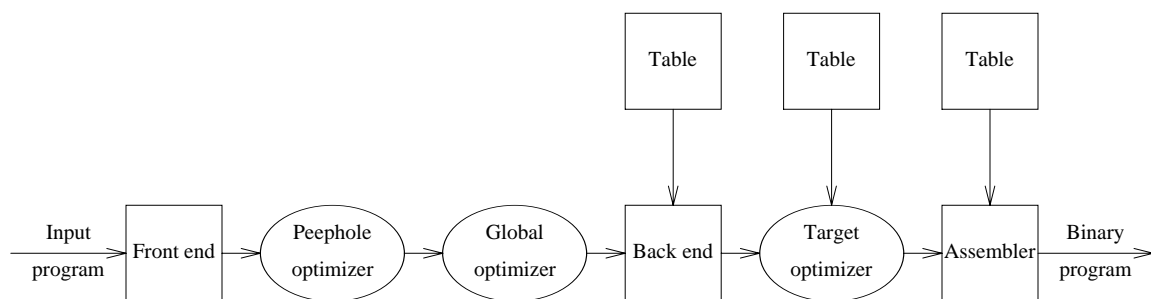


Fig. 1. Structure of the Amsterdam Compiler Kit.

*Present address: Vakgroep Informatica, Universiteit van Amsterdam, Amsterdam

† UNIX is a Registered Trademark of AT&T Bell Laboratories.

This scheme is highly flexible. It almost completely decouples the front end from the back ends. At present front ends exist for Pascal, C, Modula-2, Basic, Orca, and Occam. Back ends exist for the Motorola 68000 and 68020, Intel 8088 (with the 80386 in progress), DEC PDP-11 and VAX, Zilog Z80 and Z8000, National Semiconductor 16032, Mostek 6502, and others. By combining a front end with a back end, we get a compiler for a given language-machine combination. Using the six languages and ten machines cited above, we can make 60 different compilers.

To implement a new language, all that is necessary is to write a front end that converts the language to the simple EM stack notation. Having done this, one automatically has optimizing compilers for 10 different machines. Similarly, when a new machine appears, one has to write tables for the back end, target optimizer, and assembler, and one suddenly has compilers for five different languages for that machine.

2. THE ORIGINAL IMPLEMENTATION

To provide the maximum flexibility, each of the six passes of Fig. 1 was written as a separate program, accepting an input and producing an output, in the style of all UNIX filters. While this program organization had no influence on the quality of the generated code, it did affect the speed of compilation, as a large amount of information had to be passed through the pipeline. To reduce the volume of traffic in the pipeline, a data compression technique was employed. Some of the time gained by having fewer bytes to read and write was lost due to the overhead required in compressing and decompressing the data at every stage.

A second source of overhead was the need to load up to six fairly large programs from the disk for each compilation. In an environment such as is commonly found at universities these days, with diskless workstations and central file servers, loading all the compiler passes requires substantial network I/O in addition to the disk I/O.

A third source of overhead was the code generation algorithm used by the back end. This algorithm examined sequences of incoming EM instructions and matched them against a large and complex table, looking for the best matches. In many cases multiple matches can be found (corresponding to different code sequences), so searching multiple paths is necessary to find the best one.

All these factors, plus the presence of three optimizers clearly biased the system in favor of producing good code at the expense of fast compilation. Nevertheless, we eventually came to realize that for many applications, such as debugging and student programming labs, compile speed is much more important than execution speed. This realization led to an alternative way to implement the pipeline.

3. THE DESIGN FOR FAST COMPILATION

Two things were clear from the beginning of the redesign. First, a simpler but faster code generation scheme was needed, even if this meant sacrificing some execution efficiency. Second, the modularity provided by having independent programs for each pass was an expensive luxury we could not afford.

Solving the first problem was relatively straightforward. Instead of matching sequences of EM instructions against sequences in the back end's tables, we decided to treat each EM instruction as a simple macro to be expanded into a fixed pattern of target machine instructions (with parameter substitution). Conceptually, when the backend reads an incoming EM instruction, it uses the opcode number as an index into a table to find the corresponding byte string to output. (The actual implementation is different, as described below, but this table-lookup gives the basic idea.)

Tackling the second problem was also easy. All the existing front ends have been designed in a structurally similar way. When a front end wants to generate a particular EM instruction, say XXX, it does not just call the C function *printf*, but instead calls *gen_xxx* which handles data compression internally and then does the actual output. What we have done is replace all the *gen_xxx* routines with a different set (i.e., a different library), in which the new routines do the macro expansion directly and output relocatable binary code. In this way the entire compiler is reduced to a single program, eliminating all the pipes and intermediate data streams.

4. IMPLEMENTATION OF THE FAST COMPILERS

A new-style back end is based on two tables. The first one gives for each EM instruction its expansion into symbolic (i.e., ASCII) assembly language. The second table gives the binary output for each assembly language instruction. This approach is much simpler for the compiler writer than having to specify the direct mapping from EM instructions to relocatable binary. After both tables have been written, the ACK tools combine them so that there is, in fact, a direct mapping between EM instructions and relocatable binary.

The result of combining these two tables is a set of C functions of the form *gen_xxx*, one per EM instruction, which when called, directly output the proper relocatable binary code onto the output file (Kaashoek and Langendoen, 1988). The next step consists of linking the front end to these routines to produce an executable binary of the compiler. This binary contains the front end, which makes calls on routines that directly produce relocatable binary code for the relevant target machine. Thus the combination of the two tables is done once—at the time the the compiler itself is being produced. It is not done when individual programs are being compiled.

In addition to producing the EM to assembly language and assembly language to binary tables, the compiler writer has to provide a few C routines, such as a routine that generates the code for procedure entry (setting up the stack frame, etc.). Also required are the definitions of bytes, words, longs, pointers, segments, symbols, alignment, and other such items to customize the compiler to a particular target machine.

Although the goal of this design was to minimize compilation time without regard to object code quality (after all, the original compilers were still available), it soon became apparent that a small amount of optimization could greatly improve the object code quality at virtually no penalty in compile speed. Consider, for example, the assignment statement $i = j + k$;, where i , j , and k are 4-byte integer variables in memory. The EM code produced is:

```
lol j           / push j onto the stack
lol k           / push k onto the stack
adi 4           / add the two 4-byte integers on the stack
stl i           / pop a word from the stack and store it in i
```

Now consider the code that would normally be produced from this EM code using the most straightforward macro expansion. This example is for a typical register machine using a scratch register *r1* and a stack pointer, *sp*. The notation "move a,b" means move a to b.

```
push j         / push j onto the stack (expansion of lol j)
push k         / push k onto the stack (expansion of lol k)
pop r1         / fetch k (expansion of adi 4)
add (sp)+,r1  / add j to k (expansion of adi 4)
push r1        / push the sum (expansion of adi 4)
pop i          / store the sum in i (expansion of stl i)
```

The optimization that immediately springs to mind is the replacement of push/pop sequences by a single instruction. Applying this optimization we get:

```
push j          / push j onto the stack
move k,r1       / move k to r1
add (sp)+,r1    / add j to r1
move r1,i       / move the sum to i
```

This code is not optimal, of course, but it is only one instruction worse than the best possible code for register machines, which is

```
move j,r1       / move j to r1
add k,r1        / add k to r1
move r1,i       / store the sum in i
```

Thus by applying a very simple peephole optimization to the straightforward macro expansion, we produce code that is only slightly worse than the best possible code. Since the additional compilation time needed to check for a small number of optimizations of this kind is negligible, we have provided a mechanism for the compiler writer to specify this kind of peephole optimization. Furthermore, most of the overhead in performing the optimization is recouped in having fewer bytes to write to the output file.

Two points are worth emphasizing. First, the fast back ends directly produce relocatable object code instead of symbolic (ASCII) assembly code that must be processed by a separate assembler. This approach gains a considerable amount of performance by not having to load the assembler and rescan the input.

Second, making back ends in this form is much easier than making a conventional back end. Writing roughly 100 macros takes typically two weeks, whereas writing a full-blown optimizing back end table can take as much as six months.

5. PERFORMANCE OF THE RESULTING COMPILERS

The old saying "The proof of the pudding is in the eating" also applies to compilers. To see how fast our compilers were, we ran extensive benchmarks on the Sun 3/50 and the VAX 11/750, comparing our new compilers with the standard UNIX C compilers on those machines. We could have equally well done the benchmarking in Modula-2, or another language, but we lacked universally accepted compilers to use as a reference.

The benchmarks consisted of compiling a number of the standard UNIX utility programs using the standard UNIX compiler, *cc*, and our new fast compiler, *fcc*. The tests were made in two ways, pure compilation (*cc -c prog.c*) and compilation plus linking (*cc prog.c*). In the latter case, the pure differences in compiler speed are diluted by the addition of the link phase. The *cc -c prog.c* test is probably the more important of the two, however, since in programming projects using *make* there are typically many compilations for each call to the linker.

The data with the program sizes (number of bytes and number of lines), and the compilation times (in seconds) for the Sun 3/50 and VAX 11/750 are given in Fig. 2. The sizes are those after the C preprocessor has been run, so they differ slightly from the Sun to the VAX due to the different header files. In Fig. 3 the *fcc/cc* ratios as a function of program size are plotted. These curves are derived from the data presented in Fig. 2.

Compile times Sun 3/50						
file	# bytes	# lines	fcc	cc	fcc -c	cc -c
null.c	11	3	2.1	5.1	0.2	3.9
time.c	1104	60	4.3	9.4	0.9	6.1
cmp.c	1713	122	4.0	9.3	0.8	6.4
cal.c	2656	204	3.8	10.0	0.7	6.8
cat.c	3949	223	4.8	11.2	1.2	6.8
cp.c	4800	242	5.4	13.1	1.7	8.1
mv.c	5885	311	6.4	11.2	2.0	8.8
grep.c	7565	524	5.0	11.3	1.8	8.7
dd.c	10809	610	5.3	13.2	2.4	10.7
tc.c	11739	637	5.9	16.2	3.0	13.6
ls.c	14406	701	9.6	19.4	3.8	14.4
od.c	16805	883	6.2	15.8	3.0	14.7
ctags.c	19856	1029	6.5	21.6	3.7	16.9
ed.c	24850	1762	9.6	25.4	5.3	22.5
tar.c	26861	1417	10.8	26.2	6.8	21.3
sccs.c	31956	1557	9.8	19.8	3.9	15.8

Compile times VAX 11/750						
file	# bytes	# lines	fcc	cc	fcc -c	cc -c
null.c	11	3	3.8	5.5	0.9	3.0
time.c	1104	60	7.3	10.5	3.1	6.3
cmp.c	1713	122	5.5	12.3	2.8	7.9
cal.c	2656	204	5.1	11.5	2.6	9.3
cat.c	3949	223	9.6	17.5	5.8	16.0
cp.c	4800	242	9.4	19.2	6.1	15.1
mv.c	5885	311	12.3	19.1	9.2	16.1
grep.c	7565	524	11.9	20.2	9.0	17.2
dd.c	10809	610	14.6	24.9	9.9	22.9
tc.c	11739	637	20.5	48.1	17.7	47.5
ls.c	14406	701	19.7	41.8	14.1	38.5
od.c	16805	883	17.0	35.9	14.5	33.9
ctags.c	19856	1029	17.1	42.9	13.4	37.9
ed.c	24850	1762	26.2	63.0	21.9	58.1
tar.c	26861	1417	27.9	64.7	23.4	56.9
sccs.c	31956	1557	22.8	45.8	15.9	42.9

Fig. 2. Program sizes and compilation times on the Sun 3/50 and VAX 11/750.

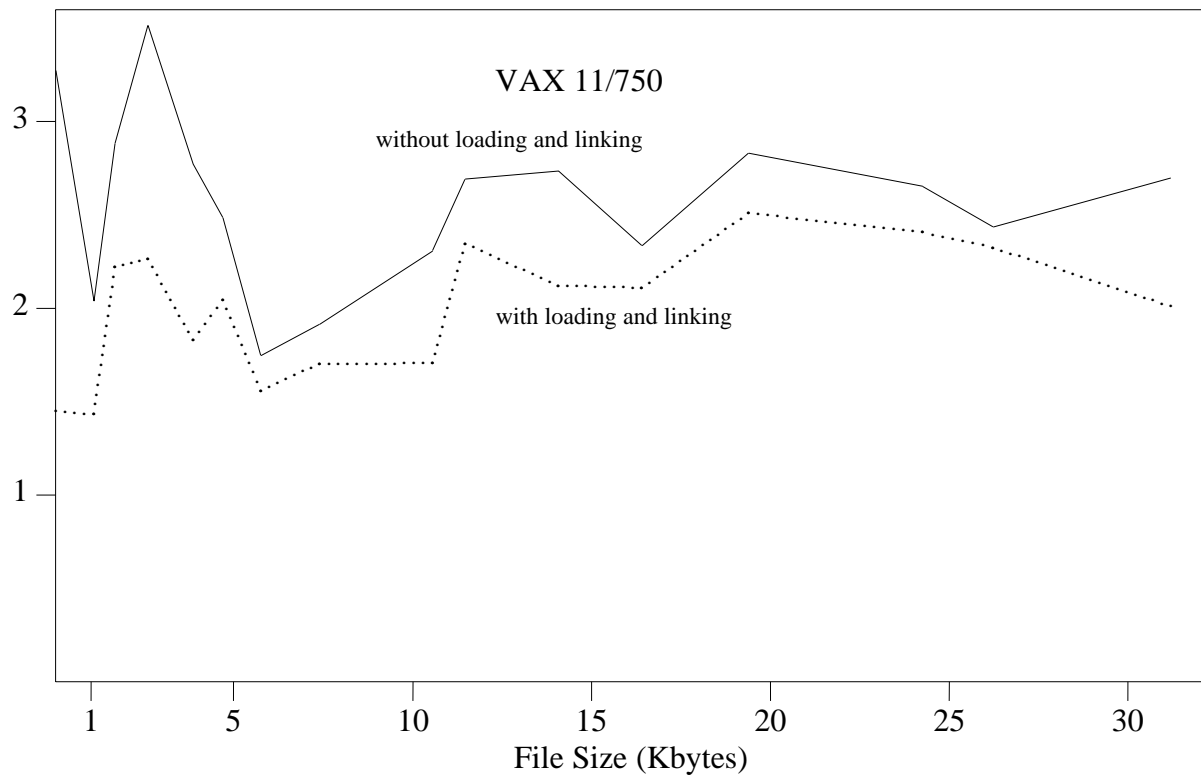
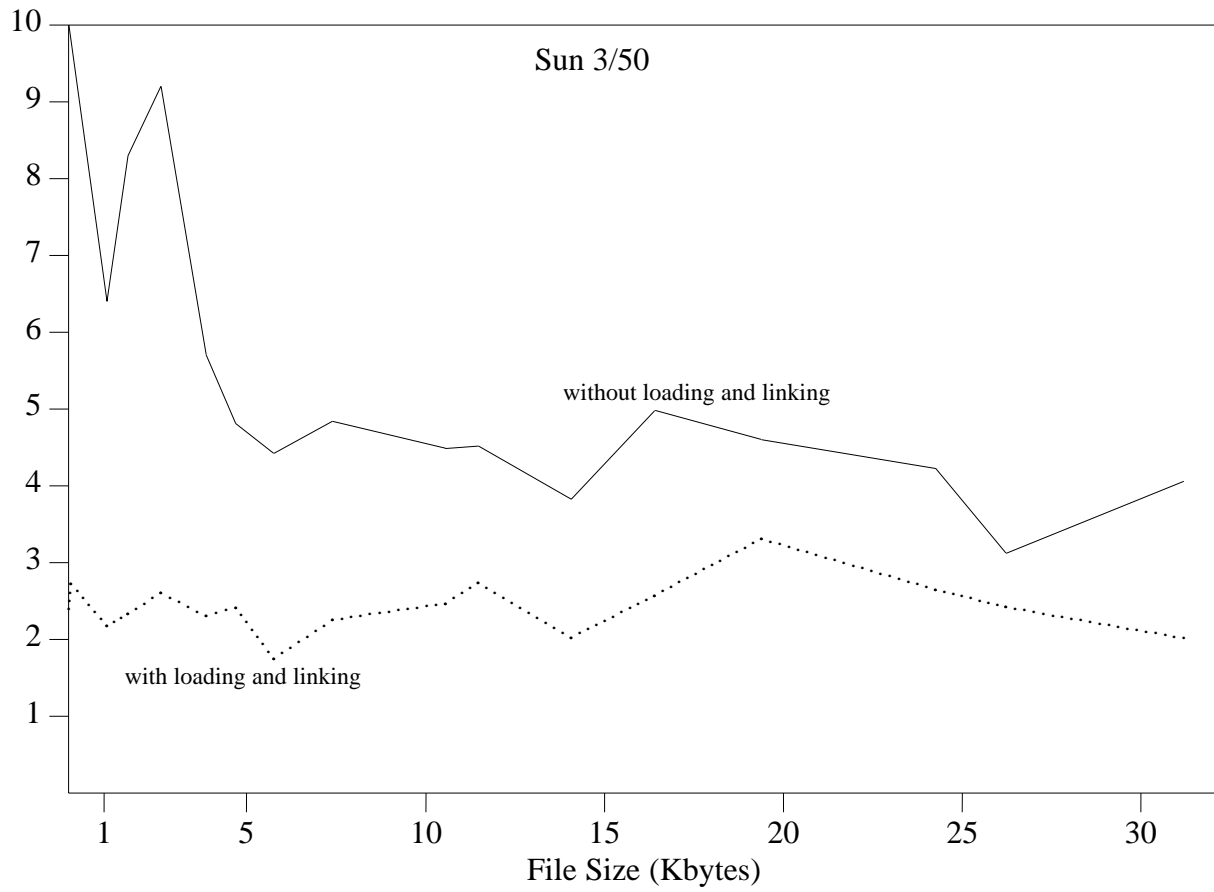


Fig. 3. Ratio of compile times fcc/cc on the Sun 3/50 and VAX 11/750.

6. DISCUSSION

From Fig. 3 we see that on the Sun, *fcc -c* is 5 to 10 times faster than *cc -c* for files smaller than 5K, and about 4 times faster for larger files. For the VAX, the speedup is about 2 to 3 times, independent of the file size.

The difference between the Sun and VAX is primarily due to the fact that the VAX compiler is a relatively faster compiler. The effect of program size on compilation speed is related to the time it takes to load the compiler. For very small programs, the dominant effect is how big the compiler, assembler, and linker are, rather than how fast they are. Most of the time is spent loading them, rather than running them. The *fcc* code is a single program that reads the source and produces relocatable *.o* files. It is considerably smaller than the Sun or VAX compiler + assembler. Both *fcc* and *cc* use the same linker, so that cancels out.

It is clear from the measurements that eliminating all the passes and the heuristic search for optimal code has led to a major improvement in compile speed. Due to the structure of ACK, the faster code generation applies not only to C, but to Modula-2 and the other languages for which a front end exists, so the modularity and portability have not been lost.

The code quality is not nearly as good with the fast compiler, but that problem can easily be finessed. While debugging a program, one uses the fast compiler, since execution speed is irrelevant until the program has been at least moderately debugged. Once that point has been reached, the normal compiler can be used to produce a *.o* file with good execution speed. For programs consisting of multiple files, work can then begin on the next file, again using the fast compiler. The *.o* files produced by the normal and fast compilers are compatible, so that during development, there is no problem mixing a collection of files, the debugged ones using the normal compiler and the ones being worked on produced by the fast compiler.

When the entire program has been debugged, the global optimizer (Bal and Tanenbaum, 1986) can be used to produce extremely high quality code. Thus by choosing among fast compilation, normal compilation, and global optimization, one can have the benefits of fast compilation during debugging and fast execution during production.

The fast compilers for C and Modula-2 are now available. Interested parties should contact the authors (by email, if possible) for more information.

ACKNOWLEDGEMENTS

We would like to thank Henri Bal and Dick Grune for their help, suggestions, and general advice concerning this project.

REFERENCES

- Bal, H.E. and Tanenbaum, A.S.: "Language and Machine-Independent Global Optimization on Intermediate Code," *Computer Languages* vol. 11, pp. 105-121, 1986.
- Kaashoek, F., and Langendoen, K.: "The Code Expander Generator," Dept. of Math. and Computer Science Report IM-9, Vrije Universiteit, 1988.
- Tanenbaum, A.S., van Staveren, H., Keizer, E.G., and Stevenson, J.W.: "A Practical Toolkit for Making Portable Compilers," *Commun. ACM*, vol. 26, pp. 654-660, Sept. 1983.