Barking Up The Wrong Tree: Why Optimizing Compilers Are Still Unable To Match Assembly Language

Marc Sabatella
University of California, Berkeley

August 2, 1988

Abstract

People often write in assembly language because such programs tend to run faster than equivalent programs written in a higher level language. Through a case study of a particular program in both assembly language and C versions, we identify some of the factors that contribute to the superior performance of the assembly language version. Much of the disparity may be attributed to interprocedural register usage. Current global optimizers do not address these issues. A class of interprocedural optimizations which we term *galactic optimizations* is introduced in an attempt to mimic the techniques employed by assembly language programmers. The optimizations are *parameter mode strength reduction* (PMSR), *global variables* (GV), and *pass in register* (PREG). The implementation of the optimizations may be at link time, or else separate compilation may be precluded.

1 Introduction

There are many reasons to choose a high level language over assembly language for the development of a software project. Many people claim that programs written in high level languages are easier to develop, debug, and maintain, and are easier to read than programs written in assembly language. While some may debate these points, it is certainly the case that one can write portable programs more easily in a high level language than in assembly. A program written in a high level language can, if properly coded, be compiled and executed with little or no alteration on widely different systems and architectures. An assembly language program, on the other hand, is bound to the system for which it was developed.

There are two types of situations in which assembly language has been traditionally preferred. The first is for systems programming, where the programmer must be able to work at the hardware interface level. However, with the widespread availability of systems programming languages such as C, this has become less of a concern. The second common reason for writing in assembly language is to produce efficient programs. Efficiency may be measured in terms of space usage, but most commonly it is measured in execution speed. In either case, even when optimizing compilers are used, a program written in a high level language will never be as efficient as an equivalent hand coded assembly language program.

It is our contention that current optimizing technology does not address the underlying reasons for the superiority of assembly language in this regard. The types of optimizations normally performed by modern compilers do not reflect techniques used by assembly language programmers. Through a case study of a typical assembly language program and an equivalent C program, we identify some of the major factors that account for the greater efficiency of the assembly language version. We then introduce a class of interprocedural optimizations which were developed as a natural implementation of the techniques ordinarily employed by assembly language programmers, and measure the effect of these optimizations on the performance of the C program. We call these *galactic optimizations*, because they represent a step beyond local and global optimizations.

2 The Environment

It was important for the purposes of this project that we choose a machine and operating system that are in widespread use, as we wish our results to be representative of a large number of programs. The system chosen was an MS-DOS personal computer. Some of the results we obtain may be specific to this system, but we believe most will generalize.

2.1 The Processor

The machine is based on a 4.77/10 Mhz (switchable) Intel 8088 microprocessor. The 8088 is a 16-bit processor with an 8-bit data path, and a 1 megabyte address space. The features of this processor that are especially relevant to code generation are the relatively small number of general purpose registers (six 16-bit registers) and their semidedicated nature. The four "scratch pad" registers are called AX, BX, CX, and DX, and each can optionally be referenced as two 8-bit registers. Some instructions will only work with specific X registers. There are two index registers, SI and DI, which may be used as general purpose registers, but they have specific functions as well. There are, in addition, two stack registers (a stack pointer and a frame pointer) which may only participate in a limited number of operations, and four segment registers, which aid in the relocation

	Compile Time (in seconds)	Execute Time (in seconds)
Turbo C	21.92	93
Datalight C	21.53	94
Optimum C	56.19	74

Figure 1: Dhrystone benchmark timings.

of code and the separation of code, data, and stack areas of memory.

2.2 The Operating System

MS-DOS is a single user, non-multitasking operating system marketed by Microsoft for personal computers. From a user interface standpoint, it is similar to Unix, and the command interpreter is similar in principle to the various Unix shells. For most purposes, MS-DOS limits the user RAM address space to 640K. The particular configuration we used was 640K user memory with 128K or 256K reserved for RAM disk. The system supports a hardware timer with resolution to hundredths of a second. Our system also contained a 20MB Winchester drive.

2.3 The Compiler

There were actually two compilers used in this project, as well as the assembler. Both are compatible with the standard Intel object module format, so routines written in C may be freely mixed with assembly language routines.

Turbo C The primary compiler used was Turbo C by Borland International. This is a relatively new compiler, although Borland has marketed a Pascal compiler for several years. Turbo C has become popular due to its friendly user interface and fast compile time. The relevant feature for this project is that it produces excellent code based on global register allocation techniques. It is capable of producing assembly language source as output that is directly compatible with the MS-DOS assembler.

Optimum C Datalight's Optimum C is a relatively unknown compiler, but it has consistently beaten all other MS-DOS C compilers in benchmark comparisons over the last two years. When global optimization is disabled (the compiler in this configuration is known as Datalight C), it performs similarly to Turbo C, both in compile time and execution time, on standard test suites such as the Dhrystone (Figure 1).

The compiler operates in two passes, and the optional global optimizer may be interposed (the compiler in this configuration is known as Optimum C). The optimizer implements virtually all of the optimizations discussed in textbooks, such as dead code elimination, global common subexpression elimination, etc. On contrived examples like Dhrystone, global optimization appears to be a big win.

TCDEBUG We used a public domain source level debugger and execution profiler called TCDEBUG as an aid in obtaining our results. The profiler divides a program into a collection of bins, and periodically during program execution, it records the location of the program counter (PC). It then is able to report on the relative percentage of time spent in each bin. This can be used to find which functions are dominating the execution time of a program, and it can also give a measure of the ratio of computation to I/O.

3 The Case Study

In keeping with our desire to be representative of actual programs, we chose a real world program rather than a standard benchmark program for our case study.

3.1 ADA_SCAN

DASC is a parser for the Ada language written by Robert Dewar in 8088 assembly language for the MS-DOS operating system. The lexical scanner, ADA_SCAN, was hand-coded in assembly language for the sole purpose of improving execution speed. While lexical analysis is often considered to be an I/O-bound task, Dewar's ADA_SCAN includes an efficient block input routine, and does all of its processing from a 36K input buffer. As a result, the scanner is highly computation intensive. The lexical analysis is performed by an ad hoc table driven algorithm which closely resembles predictive parsing. There is one function for almost every type of Ada token, and a driver that determines which function to call based on the first character of a lexeme. The individual functions then attempt to scan out the rest of the expected token.

The equivalence of programs written in two different languages is not necessarily a well defined concept. For the purposes of making meaningful comparisons with the assembly language version, we decided to perform a function by function transliteration, keeping the overall structure of the program. Within a function, structured conditionals and looping control constructs were used wherever possible, but in some cases the "spaghetti code" in the original could not be simplified without substantially changing the appearance of the algorithm. In these few cases, unconditional branches (the infamous **goto**) were used. Occasionally, ADA_SCAN makes use of hardware dependent code. For example, the routines to scan identifiers and comments used the XLAT and

	Execution Time (in seconds)
ASM	3.02
Turbo C w/**fgets**	15.98
Turbo C	6.70
Datalight C	6.98
Optimum C	6.51

Figure 2: Execution times for ADA_SCAN

XCHG instructions within a loop to pack two characters into a single register and perform table lookup two characters at a time. The control structure necessitated by this usage included a loop with multiple entry and exit points. In order to insure portability in these cases (and to simplify the control structure), the algorithm was modified to use a simple loop which scans characters one at a time.

Testing was performed on both physical and RAM disks. Physical disk timings are difficult to verify because they depend on the disk organization and can be affected by input file fragmentation. They are included because they represent a common development environment. DASC was actually designed to be used as a syntax checker as part of an integrated editor, so the RAM disk represents the intended use of the program on source files that reside completely in core.

3.2 The Benchmark

For the purposes of benchmarking performance, a main routine was written in C which does nothing but open the input file and call the lexical scanner to scan tokens repeatedly in a tight loop. The original block input routine was kept intact and was used unchanged for both the assembly language and the C versions of ADA_SCAN. The main routine and input routine were linked with the appropriate version of the scanner. Actual Ada programs were used as input. Due to timing irregularities at 10 Mhz, all tests were performed at 4.77 Mhz. To avoid excessive I/O overhead, the tests were run from RAM disk.

3.3 The Results

The first test was performed using **fgets** for input in the C version rather than the block routine (see Figure 2). This represents the way many C programmers would have approached the task. Turbo C was used for this test.

This version was indeed slower than the assembly language version, by a factor of

Code Sizes (in bytes)		
Source:		
	ASM	82570
	C	35388
Object:		
	ASM	8224
	C	14365
Executable:		
	ASM	50892
	C	52908

Figure 3: ADA_SCAN **benchmark comparative size statistics.**

over five. The implementation of **fgets** in the Turbo C library uses buffered I/O, but the block size is relatively small, so the resultant program is heavily I/O bound.

Even when the C version is adjusted to use the block input routine, it is still about 80% slower than assembly language. The global optimizations performed by Optimum C caused only a minor improvement over unoptimized Datalight C. Turbo C, which performs some global optimizations, placed between these two.

Figure 3 shows some size comparisons. The source size comparisons are dominated by the length of comments, and should not be taken too seriously. The difference in object module is more significant. Notice that the executable image size difference is minimal; these figures are dominated by the overhead of the C startup routine and the portion of the standard library used in the main routine.

4 The Analysis

The problem with global optimization is that it does not reflect the techniques actually used by to the assembly language programmer. Optimizations such as induction variable and loop invariant removal, documented in sources such as [1], which can be implemented in high level source code are no more likely to be performed by the assembly language programmer than by the C programmer. A programmer who writes an invariant expression within a loop in C is also likely to do so in assembly language; and inversely, one who identifies and removes such expressions in assembly language will probably do so in C as well.

In order to discover some of the real reasons that assembly language programs run so much faster, we switched to Turbo C and examined the assembly language output.

	Execution Time (in seconds)
ASM	3.02
C	6.70
PMSR	6.04
GV	5.16
PREG	4.23
DF	3.44

Figure 4: Execution times for ADA_SCAN: the effects of galactic optimization

Most of the differences between the hand coded and the compiler generated assembly language involved register usage. The 8086 has a limited number of registers, so the compiler tries to be as stingy as possible. The classic model of procedure call is adhered to strictly: arguments are pushed onto the stack in reverse order, two registers are made available within a function for local variables, and the rest of the registers are reserved for expression evaluation. Since the expressions in this program are relatively simple, often there are registers left unused, while variables are relegated to core.

The hand coded version made fairly exhaustive use of the registers. These uses seemed to fall into three basic categories: parameter passing, global variables, and temporaries. The latter category is the only one produced by the C compiler. As a result of this observation, we developed a set of optimizations and applied them to the C program. The first two optimizations were implemented as source transformations while the last was performed by hand on the generated assembly language. The results obtained by applying these optimizations are shown in Figure 4.

The issue of implementing these optimizations in a compiler is not considered here. We chose to overlook (for now) the difficulties of interprocedural data flow analysis and the problem of separate compilation. The purpose of this project is to demonstrate the desirability of these optimizations, not their feasibility. We hope that, in doing so, we will spur further research in the area. Some relevant work can be found in [3] and [4].

4.1 Parameter Mode Strength Reduction (PMSR)

In the Ada programming language, a parameter to a procedure may be **in**, **out**, or **in out**. That is, it may supply input to the procedure, deliver output, or be updated. Functions return a single value. If more than one return value is needed, then **out** parameters are used. Call by value is used for input parameters, copy out for output, and copy in/copy out for update parameters.

The C language defines only one parameter passing method; namely, call by value.

Call by reference is implemented by explicitly passing a pointer to the desired argument. Referencing a parameter passed by reference is more expensive than for one passed by value, because it involves indirection. Call by reference is used to implement both **out** and **in out** parameters.

The hand coded assembly language version of ADA_SCAN never passes a parameter by reference. Parameters that are used for output are placed in registers, as is the return value; thus, in effect, we have multiple valued functions. Update parameters are implemented by copy in/copy out. We were able to simulate this behavior in C with some simple transformations.

First, functions that were declared to be **void** (that is, they return no value) and take a reference parameter were transformed into functions returning the type of that parameter and accepting a value parameter of the same type. This has the effect of changing call by reference to call by value/result when update parameters were used. For example,

```
void foo (int *pi)
{
        *pi += 1;
}
```

would be transformed into

```
int foo (int i)
{
        return(i + 1)
}
```

and the call

```
foo(&i);
```

would be changed into

```
i = foo(i);
```

If the parameter is referenced several times in the function body, then the transformation yields a more efficient program.

In some cases, the parameter passed by reference was never used for input; only for output. In these situations, the input parameter could then be eliminated. If the function in question took a reference parameter, but already returned a value (or returned no value, but had more than one reference parameter), a different tactic was used. Turbo C supports the direct assignment to and from registers, using reserved global variables such as _AX, etc. By copying the desired output result for each parameter into a separate register, and copying that value back into the actual argument at the point of call, we were able to imitate the effects of copy in/copy out in a multiple value return setting. For example,

```
int foo (int *pi1, *pi2)
{
        *pi1 += 1;
        *pi2 += 2;
        return(3);
}
...
n = foo(&a,&b);
```

could be transformed into

```
int foo (int p1, p2)
{
        _BX = p1 + 1;
        _CX = p2 + 2;
        return(3);
}
...
n = foo(a,b);
a = _BX;
b = _CX;
```

These optimizations we call *parameter mode strength reduction (PMSR)* because they reduce the complexity of parameter referencing by removing a level of indirection and using registers. The savings introduced in ADA_SCAN was on the order of 10%.

When aliasing is considered, the semantics of copy in/copy out differ from pass by reference; in this program, it was clear that aliasing was not a problem. In order to implement this optimization in a C compiler, detailed interprocedural data flow analysis would be necessary.

4.2 Global Variables (GV)

Most compilers store global variables in memory. There is no reason why the well known register allocation algorithms could not be applied to global variables, but this is not often done because the interprocedural data flow analysis required is more complex than ordinary DFA, and the necessity of separate compilation calls for compromises. While there have been recent attempts at doing global register allocation at link time (especially Wall's [3]), it is not a common optimization.

In the case of ADA_SCAN, there is one global variable that is always kept in a register by the hand coded assembly language version. The C language version is spread out among four source files (two code, two header), and this one variable only dominates one of the files. Graph coloring algorithms applied to the files separately would produce incompatible results. However, if we make it the programmer's responsibility to choose which variables to keep in registers, then the compiler can easily honor the request.

By using the _SI reserved global variable in place of the global variable **source_line** (which is a pointer into the text buffer), we were able to achieve this in the C version of ADA_SCAN. For example, the code sequence

```
if (*source_line == '=')
{
        ++source_line;
        return(TLEQ + 256*CRELOP);
}
```

became

```
if (*(char *)_SI == '=')
{
        ++_SI;
        return(TLEQ + 256*CRELOP);
}
```

Because this variable was so heavily used, making this simple change improved execution time by about 15%.

4.3 Pass By Register (PREG)

This is the simplest optimization, and is in fact performed by several compilers for other machines. Rather than pass arguments on the stack, we reserve N registers, and pass the first N arguments in these. For update parameters, we make sure the input is through the same register chosen by PMSR for output; thus the parameter may reside in the register throughout the function call. If parameters to a function are passed in this manner, the stack under the return address is clean, which means we can trivially implement tail recursion elimination (TRE). TRE is almost a necessity for LISP compilers, where recursion overhead costs can otherwise be prohibitive, but is often not performed in C compilers. Keeping in mind that tail recursion elimination applies not only to truly recursive functions, but to any function whose last executable statement (other than a **return**) is a function call, TRE can be an effective optimization even in a language that does not depend heavily on recursion.

Pass by register and tail recursion elimination were implemented by hand on the compiler generated assembly language code. Whenever possible, register assignments were made in a manner compatible with the assignments made by PMSR. For example, after PMSR and GV, the function **scan_kwd** looked like this:

```
TOKEN scan_kwd (TOKEN tok)
{
      if (id_table[*(char *)_SI] == 0x00)
            return(tok);
      return(scan_id());
}
```

and the compiler generated assembly language was

```
_scan_kwd     proc    near
      push    bp
      mov     bp,sp
      mov     bx,si
      mov     al,BYTE PTR [bx]
      mov     ah,0
      mov     bx,ax
      cmp     BYTE PTR _id_table[bx],0
      jne     @229
      mov     ax,WORD PTR [bp+4]
      jmp     SHORT @228
@229:
      call    _scan_id
      mov     sp,bp
@228:
      pop     bp
      ret
_scan_kwd     endp
```

After performing these PR and TRE, we had

```
_scan_kwd     proc    near
      mov     bx,al
      mov     cl,BYTE PTR [bx]
      mov     ch,0
      mov     bx,cx
      cmp     BYTE PTR _id_table[bx],0
      jne     @229
```

```
        jmp     SHORT @228
@229:
        jmp     _scan_id
@228:
        ret
_scan_kwd       endp
```

The resultant program ran over 15% faster.

4.4 Dominant Functions (DF)

At this point, we have improved the performance of the C version of ADA_SCAN to within about 30% of that of the assembly language version, down from about 80%. If these optimizations can be automated, then many of the reasons for writing in assembly language can be circumvented. When extreme efficiency is necessary, however, the 30% difference is significant. But rather than develop the whole system in assembly language, it might make sense to write most of the code in C, or some other high level language, and use assembly language for a few critical routines. An execution profiler then becomes an extremely useful commodity.

The profiler we used partitions a program into a number of bins, and on every real time clock tick, records the position of the PC. At the end of the execution, it reports on the percentage of time spend in each bin. We chose appropriate parameters so that the bins would closely correspond with individual functions, and as a result we were able to obtain a good estimate of the relative amount of processing time consumed by each function.

Figure 5 shows the dominant functions in the assembly language, Turbo C, and optimized Turbo C versions. It should be noted here that the ADA_SCAN benchmark program consists of over 50 functions. **ada_scan** is the entry point to the scanner and is called on each iteration of the main loop. **scan_sp** scans white space characters, **scan_id** scans identifiers, and **scan_mn** scans the minus sign (used in Ada to begin a comment). It is clear that **scan_id** and **scan_mn** dominate the C versions to an extreme that is disproportional to the assembly language version. These routines, not coincidentally, featured some of the most contorted and machine dependent code in the hand coded assembly language program.

By replacing these two routines alone with their original assembly language versions, we observed a further 15% improvement in execution speed. This brought the program to within 15% of the pure assembly language version. Thus, if the optimizations described previously are available, a programmer can obtain excellent results by developing a system primarily in a high level language, and resorting to assembly language in only a few critical routines.

ASM	
ada_scan	36.6%
scan_sp	11.0%
main	8.7%
scan_id	7.9%
Turbo C	
scan_id	23.4%
scan_sp	14.5%
scan_mn	13.5%
ada_scan	13.4%
Optimized Turbo C	
scan_id	35.8%
scan_mn	13.9%
ada_scan	9.8%
main	8.4%
scan_sp	8.2%

Figure 5: Hot spots: percentage time consumed by various routines.

5 Conclusions

Most of the conclusions we reached have been described throughout this paper. By examining a real world program, we were able to identify some differences between assembly language and high level languages that account for the better performance of assembly language programs. We then proposed a set of optimizations designed to emulate techniques used by assembly language programmers. We made no effort to show how these optimizations could be implemented, nor did we prove that the results would generalize to apply to arbitrary C programs. Some unanswered questions had to do with interprocedural data flow analysis and separate compilation. In short, more research will be necessary to determine if these optimizations can become a permanent part of the compiler writer's bag of tricks.

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[2] Department of Defense, *Reference Manual for the Ada Programming Language*, Springer-Verlag, New York, 1983.

[3] David W. Wall, "Global Register Allocation At Link-Time", *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, published as *SIGPLAN Notices* 21 (7): 264-275 (July 1986).

[4] David W. Wall, "Register Windows vs. Register Allocation", *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, published as *SIGPLAN Notices* 23 (7): 67-78 (July 1988).