Published in *Proceedings of the 5th International Conference on Signal Processing Applications and Technology*, pp. I-689–I-694, October 1994.

An Optimizing Compiler for the TMS320C25 DSP Chip

Wen-Yen Lin, Corinna G. Lee, and Paul Chow

Department of Electrical and Computer Engineering, University of Toronto
10 King's College Road, Toronto, Ontario M5S 1A4
CANADA

Abstract

Programming DSP applications in high-level languages such as C is becoming more prevalent as applications become increasingly more complex. Current DSP compilers, however, are generally unable to exploit the DSP-specific features of a processor to produce good codes for most DSP applications. To explore the challenges and gain an understanding of generating code that performs as well as handwritten assembly code, we developed an optimizing C compiler for Texas Instrument's TMS320C25 DSP chip. In this paper, we describe the UofT C25 optimizing compiler and show that it is able to generate code that is comparable or, for some DSP algorithms, superior in performance to code generated by TI's TMS320C2x/C5x optimizing compiler. Moreover, by utilizing specific features of the TMS320C25, the UofT C25 compiler generates code that uses only 1.5 to 2 times as many cycles as handwritten assembly code.

1.Introduction

Contemporary DSP processors typically provide special hardware features that are designed to execute DSP applications quickly [1]. For example, DSP architectures are usually designed with the Harvard structure, which has two or more memory banks for instructions and data, thus allowing simultaneous memory accesses. Another innovation is the parallel execution of multiplication and accumulation, which are core operations for many DSP applications. DSP chips also often have a special instruction that can execute the overhead operations for a loop in a single cycle. Finally, DSP chips usually provide special addressing modes for specific popular DSP algorithms. For example, modulo addressing or circular buffer addressing is often implemented for FIR filters, while bit-reversed addressing is used for FFTs.

To utilize these special hardware features, programmers historically have written DSP applications in assembly language. This, however, requires programmers to have a thorough understanding of both the target application as well as the processor architecture. For example, to use the integrated multiplier/accumulator at full speed, a programmer needs to organize instructions and data in separate banks to perform a multiplication and an accumulation simultaneously within one machine cycle. Furthermore, the programmer has to examine the instruction pipeline carefully and arrange the instruction streams properly to exploit any potential parallelism as much as possible [2].

Programming in assembly language, however, becomes increasingly difficult as DSP applications become larger and more sophisticated. Hence, programming in a high-level language such as C is becoming more prevalent because C programs are easier to code, debug, maintain, and port. Unfortunately, the performance of code generated by current high-level language DSP compilers is often poorer than that of handwritten assembly code. This is often due to the inadequate ability of the compiler to use the specialized DSP features in hardware.

To explore the challenges and to gain an understanding of generating code that is as good as handwritten assembly code, we developed an optimizing C compiler for Texas Instruments' TMS320C25 DSP chip (C25). We use the C25 chip as our target processor because it is inexpensive, it is widely used, and it has a simple architecture [3][4]. Some architectural features, such as the eight auxiliary registers, the BANZ, RPTK, and MAC instructions, and the three blocks of on-chip RAM, can be used effectively to execute program constructs such as loops and multiply/accumulate computations that occur frequently in DSP applications. Other features, most notably the use of a 3-bit auxiliary register pointer (ARP) to specify which auxiliary register to use, the lack of an index addressing mode, and an accumulator-based data path, require special attention to ensure that excessive code is not generated when accessing local variables or compiler-generated temporary variables which are stored on the program stack.

In this paper, we describe the optimizations we implemented in our compiler. We also present empirical results that compare the performance of code generated by our compiler with the performance of code generated by TI's 320C2x/C5x optimizing compiler. A performance comparison with handwritten assembly code is also presented.

2.The UofT C25 Optimizing Compiler

Our optimizing compiler consists of a modified GNU C compiler (GCC) [5] and a post-optimizer which performs optimizations specific to the C25. Figure 1 shows a block diagram of the compiler.

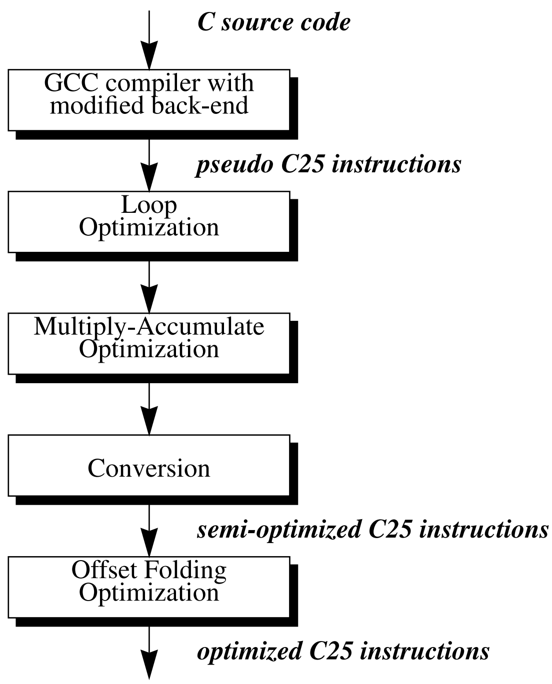

Figure 1: UofT C25 Optimizing Compiler

We created a set of pseudo C25 instructions because using addressing modes other than direct addressing often requires several C25 instructions to specify. For example, an instruction that uses indirect addressing relies on the ARP register to specify which auxiliary register contains the actual address. Hence, any operation that uses a new auxiliary register has to update the ARP register to specify the new auxiliary register before performing the actual operation. Updating the ARP register can be done either using a LARP instruction or as part of an indirectly addressed instruction. As another example, the lack of index addressing in the C25 makes accesses to the program stack very expensive and inconvenient. The addition or subtraction of an offset to the auxiliary register has to be specified and performed in a separate instruction instead of within the instruction which performs the stack access.

It is difficult to modify GCC to generate a multiple instruction sequence that uses indirect addressing or that accesses the stack. Our solution is to use a set of new pseudo instructions that directly specify the auxiliary register in the instruction that uses it and the emulated index addressing. Table 1 lists the pseudo C25 instructions that we created. These pseudo instructions are generated by GCC and are optimized and converted to real C25 instructions by the post-optimizer.

Name	Format	Conversion Example	
		Pseudo C25 instruction	Real C25 instructions
Basic indirect addressing	*op* (ar*n*)	ADD (ar1)	LARP ar1 ADD *
Pre-inc/dec addressing	*op* ++(ar*n*) *op* --(ar*n*)	ADD ++(ar1)	LARP ar1 ADRK 1 ADD *
Post-inc/dec addressing	*op* (ar*n*)++ *op* (ar*n*)--	ADD (ar1)++	LARP ar1 ADD *+
Index addressing with offset	*op* (ar*n*) + *const*. *op* (ar*n*) - *const*.	ADD (ar1)+2	LARP ar1 ADRK 2 ADD * SBRK 2
Addition of ar0 to any auxiliary register	ADDR ar0, ar*n*	ADDR ar0, ar1	LARP ar1 MAR *0+
Arithmetic operation for auxiliary register	ADRK ar*n*, *const*. SBRK ar*n*, *const*.	ADRK ar1, 10	LARP ar1 ADRK 10

Table 1: Formats of Pseudo C25 Instruction

In order to generate pseudo C25 instructions from GCC, we developed a new set of target-dependent configuration files. This set of configuration files includes a target machine definition file and an instruction pattern file. These files describe the target processor architecture which GCC is generating code for.

Some standard optimizations, such as dead code elimination and common subexpression elimination, etc., are performed by GCC. In addition to these standard optimizations, we investigated and implemented three types of post-optimizations based on the specialized C25 features. The first two optimizations, *loop optimization* and *multiply-accumulate* (MAC) *optimization*, are performed on pseudo C25 instructions, while the remaining *offset folding* optimization is performed on real C25 instructions.

The loop optimization allocates the loop index variable to a free auxiliary register instead of a memory location on the stack. The post-optimizer scans the input program for loop structures, checks that there are no other instructions reading or writing the loop index variable within the loop, and determines whether there is an auxiliary register available which is not used inside the loop body. If the candidate loop meets these conditions, then the initialization and test operations associated with the loop are substituted with a pair of LARK and BANZ instructions. The LARK instruction loads the auxiliary register assigned for the loop index variable with the repetition number, and the BANZ instruction performs a conditional jump based on the value of the auxiliary register and updates the register's value after the test. The loop optimization is able to reduce the number of pseudo instructions for a loop from $2+5n$ to $1+2n$ approximately, where n is the number of times the loop is executed.

The multiply-accumulate optimization provides fast execution for multiplying and accumulating two data arrays by using the MAC instruction. The post-optimizer analyzes the input program for instructions that successively multiply and add two data arrays and replaces these instructions by a pair of RPTK and MAC instructions. To use the RPTK instruction, the number of elements in an array must be known at compile time to perform the correct number of iterations. Since this information is already determined by the loop optimization, the MAC optimization is performed after loop optimization.

To utilize the C25 MAC instruction, one array of the operands must be stored in program memory. Hence, the post-optimizer also generates extra instructions to move one of the operand arrays from data memory to program. The MAC optimization also eliminates the original loop structure that contains the multiply and accumulate instructions if there is no other instructions within the loop. For a typical operation that multiplies and accumulates two data arrays, the MAC optimization is able to reduce the number of pseudo instructions from $7n+2$ to 9, where n is the iteration count of a loop.

The offset folding optimization eliminates and combines the selection and offset-adjustment instructions that are generated by the post-optimizer during the conversion of pseudo instructions. As we show in table 1, each pseudo C25 instruction is transformed into 2 to 3 real C25 instructions including an LARP instruction, the actual operation, and probably an offset adjustment operation. The offset folding optimization tries to eliminate the LARP by specifying the new auxiliary register in the previous indirect addressed instruction. In addition, the post-optimizer attempts to combine redundant offset adjustment instructions that arise when an auxiliary register is used twice in succession. In such a situation, two adjustment instructions will occur without an intervening indirectly addressed instruction. By keeping track of the values of the auxiliary registers as they are updated by the adjustment instructions, the post-optimizer can replace two successive adjustment instructions with one. For a pseudo instruction which uses emulated index addressing mode, the offset folding optimization reduces the number of converted real C25 instruction from 4 to 1 in the best case and 2 to 1 in the worst case.

3.Evaluation Methodology

To evaluate the quality of code generated by the UofT C25 compiler, we compared the execution time of code generated by the UofT compiler to the execution time of code generated by the TI C25 compiler [6] and to the execution time of handwritten assembly code. Figure 2 shows the experimental framework we used to evaluate our compiler. A program written in C was compiled using both the TI compiler and the UofT compiler at different levels of optimization to generate an assembly program. The assembly program was then transformed into an executable binary by the assembler and linker of the TMS320 fixed-point DSP assembly language tools from TI. The executable binary was simulated with TI's TMS320C2x simulator to obtain the execution time in machine cycles. The simulation procedure for the corresponding handwritten assembly program is identical except the compilation step is skipped. In addition, by including calls to input and output C25 assembly routines, we used the simulator to verify that UofT compiler-generated code produces correct results.

Twelve kernels and one application were used as benchmark C programs in this study. The kernels are based on six common DSP algorithms [7]:

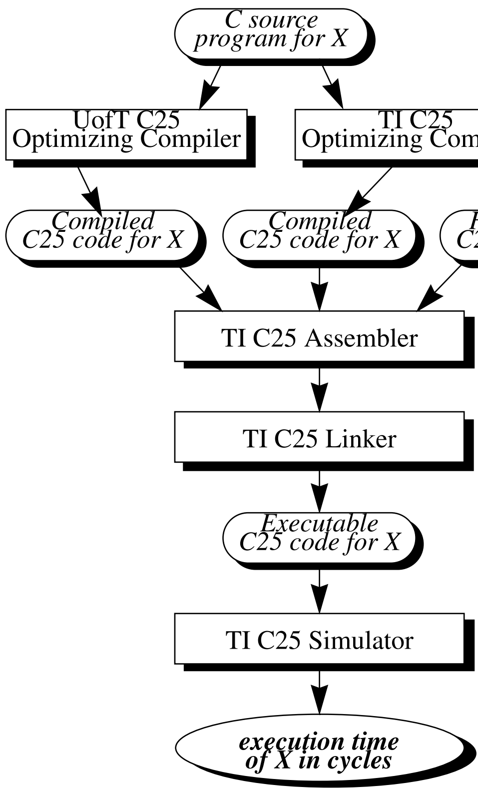

Figure 2: Evaluation Methodology

- matrix multiplication.
- finite impulse response (FIR) filter;
- infinite impulse response (IIR) biquad filter;
- least mean squared adaptive FIR filter; and
- normalized lattice filter;
- radix-2, in-place, decimation-in-time fast Fourier transform;

For each algorithm, two kernels of different problem sizes were used. A linear soft-decision decoder program for block codes [8] was also included in this study to evaluate the compiler for larger, practical DSP applications. We obtained handwritten assembly-language versions of the kernels from TI's TMS320 BBS site (*ti.com*) whenever possible and programmed the remaining kernels ourselves [9].

The TI optimizing C compiler includes four levels of optimizations from no optimization to optimization level 2. Level 0 optimization of the TI compiler performs some simple optimizations such as dead code elimination and statement simplification. Level 1 optimization performs local common subexpression elimination and local dead assignment elimination in addition to the optimizations at level 0. Level 2 optimization performs all those in level 1 plus global common subexpression elimination, global dead assignment elimination, and loop optimization.

4.Results and Discussion

Figures 3, 4, and 5 show the performance of code generated by the TI compiler, by the UofT compiler, and by a human programmer. Performance is plotted relative to the performance of the TI compiler with no optimization.

4.1. No Optimizations

Figure 3 shows the performance results for the UofT C25 compiler and the TI compiler when no optimizations are used. Code generated by the UofT C25 compiler has comparable performance to code generated by the TI compiler. Notably, code generated by the UofT C25 compiler has better performance for those kernels which have larger data arrays and more stages. The reason for this is that the TI compiler generates two test sequences for each loop construct whereas the UofT C25 compiler generates only one test sequence. The reduction in instruction count effectively provides better performance for those kernels, particularly those with nested loop structures.

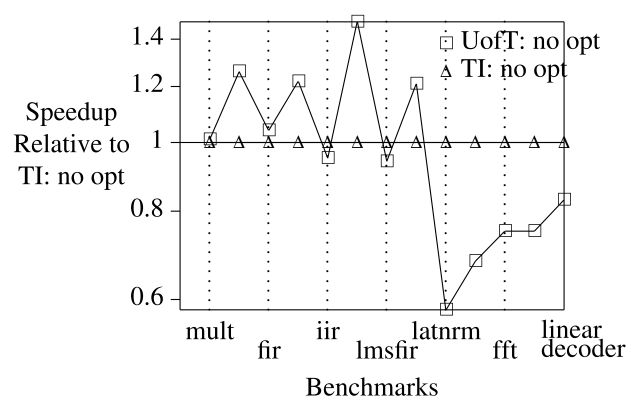

Figure 3: Performance Result for No Optimization

Unfortunately, code generated by the UofT C25 compiler has worse performance for the lattice filters, FFTs, and the linear decoder. These benchmarks use many local variables which are allocated to the stack. However, stack access is expensive in C25 because of the inefficient manipulation of auxiliary registers. The UofT C25 compiler generates extra instructions that update the frame pointer for each stack access. Some of these updating instructions can be combined together using the offset folding optimization, whose impact on performance is discussed next.

4.2. Offset Folding and Loop Optimization

Figure 4 shows the performance results for the UofT C25 compiler when the offset folding and loop optimization are used and the performance results for the TI compiler at optimization level 1. The two curves for the UofT C25 compiler represent the performances of codes generated by using loop optimization only and by using offset folding plus loop optimization. Since the empirical results indicated that code generated by the TI compiler at optimization level 0 has almost equivalent performance as code generated at optimization level 1 (the difference of average speedup is less than 2%), we will show and discuss the performance results of the TI compiler at optimization level 1 only.

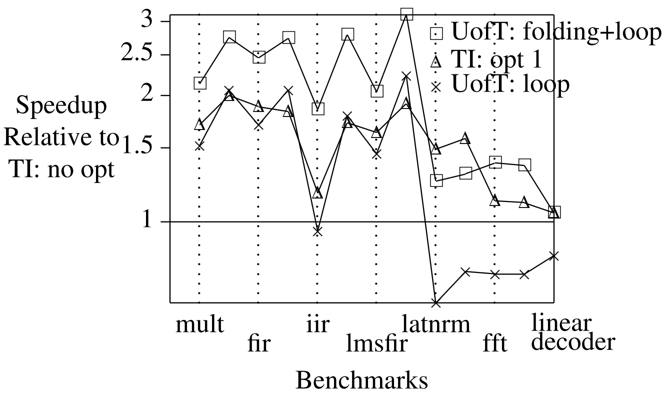

Figure 4: Performance Results for Loop Only and Loop+Folding Optimizations

Code generated by the UofT C25 compiler, when both the offset folding and loop optimization are used, has an average speedup of 2.02. In comparison, code generated by the TI compiler at optimization level 1 has an average speedup of 1.55 only. Additionally, code generated by the UofT compiler, when only the offset folding optimization is used, has an average speedup of 1.54. This result shows that the offset folding optimization not only effectively reduces the number of instructions, hence saving program memory for instruction storage, but also improves the performance of those kernels which have large data arrays and extensive loop repetitions.

Our result, however, suggests that using loop optimization only is not enough to generate high quality code. The average speedup of codes generated using only the loop optimization is 1.35. This indicates that although the loop optimization generates better instructions for a loop structure, the overall performance is not significantly improved because the offset adjustment instructions dominate the execution time of the benchmark programs.

4.3. MAC Optimization

Figure 5 shows the performance results for code generated by the UofT C25 compiler utilizing the offset folding plus the loop optimization plus the MAC optimization, the performance results for code generated by the TI compiler at optimization level 2, and the performance results for handwritten assembly code. The graph indicates that, by using the innovative MAC optimization, the UofT C25 compiler generates code that has an average speedup of 3.62, while code generated by the TI compiler at optimization level 2 has an average speedup of only 2.16. Moreover, when compared with handwritten code, code generated by the UofT C25 compiler has encouraging performance results for 8 kernels out of the total 12 kernels. Due to time constraints, handwritten assembly programs for the lattice filters were unavailable at the time of publication. The 8 kernels - the FIR filters, IIR filters, least mean squared FIR filters, and matrix multiplications - contain operations on which the MAC optimization can be applied. Hence these 8 kernels exhibit an average speedup of 5.10, which is within a factor of 2 of the average speedup of 8.91 exhibited by handwritten codes.

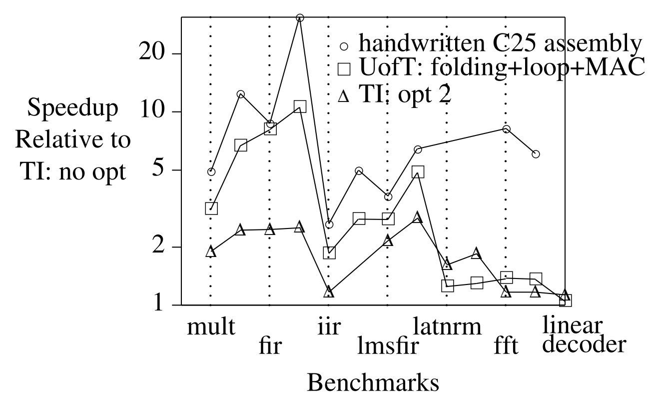

Figure 5: Performance Results for Loop+MAC+Folding Optimizations

The TI compiler, very interestingly, generates incorrect code for the 4-cascaded IIR filter processing 64 points. It allocates a local variable and a loop index variable to the same auxiliary register for a loop. However, the value of the local variable is changed within the loop and hence, the value of the loop index variable is made incorrect. We attempted to fix the bug by allocating the loop index variable to another auxiliary register. However, we could not find a free auxiliary register. Consequently. we did not include a speedup value for the large IIR filter for the TI performance curve in Figure 5.

Of the 8 kernels, the large 256-tap, 64-points FIR filter has the worst performance relative to the performance of handwritten code. The performance of this kernel could be improved by a couple of extensions of the MAC optimization. First, the post-optimizer could use the MACD instruction, which is used to implement a delayed transmission line. The MACD instruction not only performs the function of the MAC instruction but also copies a value in data memory to the next higher location. A second potential extension is to identify loop invariant code and move such code the outside the body of a loop. The large FIR filter is programmed as a double-nested loop. The current MAC optimization implements the inner loop as a RPTK/MAC pair and inserts the data-moving instructions in the outer loop just before the RPTK/MAC pair. Since the data-moving instructions are copying the filter coefficients, which never changes, these instructions could be placed outside the outer loop to be executed only once. Both extensions require data flow analysis to determine whether the extension can be applied legitimately and, hence, neither extension has been implemented yet.

Our empirical results show that the decoder application and the remaining 4 kernels - the lattice filters and the FFTs - did not benefit from the MAC optimization. The reason for this is that these benchmark programs do not contain appropriate multiply and accumulate operations, and hence the MAC optimization cannot be used on these benchmarks at all. However, the performance of FFTs could be improved by utilizing the bit-reversed addressing mode of C25. Another possible improvement could be achieved by rearranging data to use the on-chip memory more efficiently.

5.Summary and Conclusions

In this study, we have developed an optimizing C compiler for Texas Instruments' TMS320C25 DSP chip and showed that a modified GCC compiler, combined with a DSP-specific post-optimizer, is able to generate quality code. We also showed that our optimizing compiler is able to generate code that is comparable in performance to assembly code written and optimized by hand. Thus, our compiler can provide C25 application programmers with a flexible C programming environment comparable to what programmers of general-purpose processors are accustomed to.

Standard optimizations performed by current optimizing compilers for general-purpose processors are also used in most optimizing DSP compilers. Such optimizations include data flow analysis and common subexpression elimination. To fully exploit the processing power of DSP chips, however, optimizing DSP compilers must include some target-dependent and DSP-specific optimizations in addition to these standard optimizations.

The benchmark kernels used in this study represent a typical set of operations often seen in most DSP applications. They provide a good starting point to investigate and evaluate both the performance of DSP architectures and optimizing compilers. We are currently looking for larger and more practical DSP applications to gain a more thorough and comprehensive understanding of how a compiler can be used to improve the performance of DSP applications while providing the advantages of programming in a high-level language.

Acknowledgments

This research has been funded by a grant from the Information Technology Research Center, a Center of Excellence supported by Technology Ontario.

6.Reference

[1] Edward A. Lee, *Programmable DSP Architectures: Part I*, IEEE ASSP Magazine, October, 1988.

[2] Edward A. Lee, *Programmable DSP Architectures: Part II*, IEEE ASSP Magazine, January, 1989

[3] Texas Instruments, *TMS320C2x User's Guide*, 1993.

[4] Ray Weiss, *EDN's DSP-Chip Directory*, EDN, September, 1993.

[5] Richard M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, Inc., 1992.

[6] Texas Instruments, *TMS320C2x/C5x Optimizing C Compiler User's Guide*, 1991.

[7] Vijaya K. Singh, *An Optimizing C Compiler for a General Purpose DSP Architecture*, M.A.Sc. Thesis, Dept. of Electrical and Computer Engineering, University of Toronto, 1992.

[8] Stephen L. W. Ho, *DSP Implementation of a Soft-Decision Decoding Algorithm for Block Codes*, M.Eng. Thesis, Dept. of Electrical and Computer Engineering, University of Toronto, 1994.

[9] Texas Instruments, *Digital Signal Processing Applications with the TMS320 Family*, Volume 1, 1989.