Evaluation of the TMS320C6x C Compiler and Assembly Optimizer

Ranganathan Sankaralingam
Wade Schwartzkopf
Magesh Valliappan

EE382N
Superscalar Processor Architecture

The University of Texas at Austin

May 14, 1999

Abstract

We evaluate the Texas Instruments TMS320C6x C compiler and assembly optimizer. Unlike previous attempts to evaluate these tools, we measure not just execution time, but also code size, wasted program memory, and several other metrics. Also, we compare the performance of the compiler on ANSI C benchmarks with that of C benchmarks that have been optimized for the TMS3206x compiler. In addition, we examine the effects of some of the compiler options. Next, we evaluate the assembly optimizer by writing linear assembly versions of some of the ANSI C benchmarks and comparing their performance with the assembly optimizer to that of the hand-optimized and compiled C code. Finally, we also evaluate the TMS3206x C compiler on a large application, a JPEG codec.

1 Introduction

The growing demand of multimedia and digital signal processing (DSP) applications has led to the extensive use of dedicated embedded processors. Applications like video teleconferencing, digital subscriber loop systems, virtual reality, etc., require processing large amounts of data with complex algorithms. Typical DSP applications exhibit significant parallelism. Traditional DSP processors achieved speedups over general purpose processors for signal processing tasks by identifying commonly executed groups of instructions and mapping them to dedicated instructions, such as multiply accumulate, circular buffers and even Viterbi decoding (Texas Instruments' TMS320C54x). To take advantage of the parallelism, they have a Harvard architecture, which supports a limited set of parallel instructions [1].

Techniques like loop unrolling and software pipelining [2] expose more parallelism. Multiple instruction-issue can take advantage of this increased parallelism. Traditional DSP instruction sets are not a good match for multiple-issue processors, because of their rigidity and extreme specialization [3]. Two approaches to exploit parallelism are Very Long Instruction Word (VLIW) and Single Instruction Multiple Data (SIMD) architectures. Texas Instruments (TI) has adopted a VLIW approach (VelociTI architecture) while Analog Devices has chosen a static superscalar SIMD approach (TigerSHARC architecture).

In this paper, we examine an aspect of the VLIW approach. The VLIW processor has several functional units that can each execute in parallel. A single instruction packet contains multiple RISC-style operations, one for each functional unit, that the VLIW processor can fetch, decode and execute simultaneously. VLIW processors have an extremely simple control unit. It generally does not check dependencies between instructions or perform dynamic scheduling. Therefore, the burden of detecting ILP and scheduling operations onto the functional units to ensure correct program behavior falls entirely on the compiler. The performance of the compiler greatly influences the execution time and code size of the application. Because of the performance's dependence on the compiler, a study of compiler efficiency is very important.

The TMS320C6x is a VLIW DSP processor developed by Texas Instruments Inc. In this paper, we have analyzed the performance of some of the TMS320C6x development tools. The purpose of this work was to analyze the performance of the optimizing C compiler and the assembly optimizer. We wanted not only to verify the claims made by Texas Instruments about their TMS320C6x compiler, but also measure performance characteristics of the compiler not found in any previously published evaluation of this compiler. To do this, we wrote and compared

several versions of compiler/optimizer generated code with hand optimized assembly code. We then examined the code produced by the compiler and assembly optimizer to evaluate these code generation tools. Furthermore, unlike most previous attempts at evaluating this compiler, which used kernels, we also tested the compiler with a large application.

2 Texas Instruments TMS320C6x Code Generation Tools

Texas Instruments has released a set of development tools that include an advanced optimizing C compiler and an assembly optimizer. The C Compiler is ANSI compliant and accepts C source code from which it produces assembly language code. The compiler can perform several optimizations to improve the efficiency of the compiled code. The degree of optimization can be selected by setting suitable options.

The assembly optimizer is a tool designed to generate optimized parallel assembly code from linear assembly source code. Linear assembly code can be written without considering parallel instructions, latencies, and register usage. The assembly optimizer schedules the instructions for the parallel architecture to maximize throughput, taking into account the latencies and register assignment.

3 Measures of compiler /assembler performance

In the past few years, there have been several attempts to evaluate the compiler for the TMS320C6x [4, 5, 6, 7]. Each of these has used various metrics like execution time and code size to measure the performance of the TMS320C6x compiler. Below are described the factors we examined in our evaluation of the TMS320C6x code generation tools.

3.1 Execution time

Since the TMS320C6x does not perform dynamic scheduling and has a very simple control unit, execution time of code on a VLIW processor depends heavily on the extent to which the compiler takes advantage of the parallelism in the code. To achieve good performance, the compiler has to schedule instructions such that functional units are used to their maximum potential. This is directly reflected in the execution time. To expose the parallelism, the compilers use loop unrolling and software pipelining [2]. Texas Instruments claims that compiler-generated code is on average about 19% slower than hand-optimized code [4]. Compilers for traditional DSP architectures usually generate code that is 2 to 5 times slower than hand-optimized code.

3.2 Code Size

The most significant problem with VLIW architectures is the increase in code size. Since external program memory is an expensive resource in DSP applications, compiler performance in terms of code size is also very useful. However, only one previously published study of the TMS320C6x compiler looked at code size [6]. Significant factors in the size of the program code are loop unrolling and software pipelining, which are used extensively to improve performance even though they lead to significant increase in code size.

3.3 Unused Program Memory

The TMS320C6x requires that no parallel set of instruction can cross an eight-word boundary. Because of this constraint, it is sometimes necessary to place NOP's in the code so that the next instruction packet will lie on an eight-word boundary. Texas Instruments claims that these

NOP's generally account for 10-20% of total program memory [8]. We have examined each version of the compile code for every benchmark and determined this wasted program memory.

4 Evaluation Methods

We evaluate the performance of the compiler on ANSI C code, because analyzing the performance of C code that uses software pipelining and intrinsics is not necessarily a good way to judge compiler performance. One advantage of using C is that it is easier to write C code than assembly language. If software pipelining and intrinsics must be used for maximum performance, coding in C is no longer as easy. Portability also suffers, since these are processor specific optimizations.

We compared the several versions of common DSP kernels – compiler generated code from ANSI C source, compiler generated code from several version of C code optimized for the TMS320C6x, linear assembly optimizer generated code, and hand-optimized assembly code. We found and compared the execution times and code size of these DSP kernels under various conditions, such as interruptibility and the use of intrinsics in the C code. This should compare the effectiveness of the code generation tools on different types of code.

For the kernel benchmarks, we wrote three different versions of C code optimized for the TMS320C6x compiler.

- The first version was merely the ANSI C code modified by giving clues about code to the compiling. These clues include the `const` keyword and `_nassert`, which informs the compiler about how many times a loop may run. None of these modifications altered the

structure of the C code. Theoretically, any compiler for any processor could use these modifications to produce more efficient code.

- In the second version, the structure of the code was changed specifically for the TMS320C6x. Loops were unrolled, the order of lines was rearranged, and several other changes were made so that the code compiled optimally for the TMS320C6x. This code was still ANSI compatible; however, these modifications are specific to the TMS320C6x, and may not be helpful for other compilers.

- The third and final version of the C code included intrinsics, TMS320C6x machine instructions which can be called like C functions, and other things specific to the TMS320C6x compiler. This code is generally the optimal code for the TMS320C6x, but it is no longer ANSI compatible.

In addition, we also wrote linear assembly versions of some of the benchmarks to evaluate the assembly optimizer's performance. The rest of the linear assembly, the ANSI versions of C code with no optimization, and the hand-optimized assembly benchmarks were obtained from Texas Instruments' web page [9].

5 Kernel Benchmarking Results

5.1 Execution Time

Figure 1 shows a comparison of execution times of several versions of a selection of kernel benchmarks. The DCT and FFT kernels have been shown on separate scales since their speedup is much greater than the other benchmarks. These were the two kernels on which we could not extract sufficient performance from the compiler.

- Speed of each benchmark improves as more levels of optimization are applied to the C code. In all cases, the hand-optimized assembly is the fastest, but in a few cases the optimized C code is very close in its performance.

- The compiled ANSI C benchmarks often run 2-10 times slower than the hand-optimized assembly does. This means that the TMS320C6x compiler alone will not deliver efficient code - the programmer must also put an effort into helping the compiler out.

- Some improvement can be obtained by loop optimizations and other transformations of the code, retaining ANSI compatibility.

- With intrinsics we can almost match hand-optimized assembly in some of the benchmarks. However we could not achieve a speedup on some kernels even with intrinsics.

- Most of the C optimizations were obtained after numerous trials to get to the performance level shown in the figure. Optimizations are often not obvious, since it is not always clear what effect transforming the C code will have on the compiled assembly.

5.2 Code Size

Figure 2 shows the code sizes of each of the versions of the benchmarks. Again DCT and FFT are shown on a separate scale.

- In all the benchmarks, the first level of C optimization results in a code size smaller than that of the compiled ANSI C code.

- The subsequent C optimizations generally increase the code size. A noticeable exception is the IIR benchmark. Here unrolling an inner loop reduced the code size. Further analysis

revealed that the compiler did not eliminate redundant loads after unrolling the inner loop, causing an increase in the code size.

- The hand-optimized assembly versions are generally smaller than the optimized C versions. The only exception was the Block LMS benchmark, where the hand-optimized code was unrolled much more than the optimized C code.

5.3 Unused Program Memory

For each version of all of the benchmarks we have measured the amount of program memory that goes unused due to placing NOP's in the code so that instruction packets will not cross eight-word boundaries. The percentage of these NOP's found in the code is shown in Figure 3. It is difficult to detect a pattern for this value in the benchmarks and their various version. It appears to be almost a random value. However, we are able to see that this value generally lies within 10-20%, which is close to what Texas Instruments' reports in [8].

5.4 Epilog Removal

Epilog removal is achieved by setting a compiler option that forces the compiler to remove the epilog from software pipelined loops. In most cases the compiler was able to find a schedule from which the epilog could be removed. The effects of epilog removal are reduced code size and, on average, a small improvement in execution time. These effects can be seen in Figure 4.

5.5 Interruptibility

Typically, DSPs operate in asynchronous environments and use interrupts to respond to asynchronous events. So it is important to be able to generate code that can be interruptible. In the TMS320C6x, there are two primary reasons why code is not interruptible

- Pending branches in tight loops : Due to pipeline effects, this requires interruptible code to have loops longer than six cycles. This is done either by loop unrolling or inserting delays. The former would lead to an increase in code size and the latter to a drop in performance.

- Multiple register assignments : Avoiding multiple assignment places further constraints on register resources and may affect the performance by reducing the number of iterations that can execute in parallel in the software pipeline.

The TMS320C6x compiler allows code to be compiled with a given interrupt latency. By default, the TMS320C6x compiler disables interrupts around loops. Figures 5 and 6 show the ratio of the execution time and code size with the interrupt latency set at 10 cycles to the execution time and code size of the default uninterruptible code. As can be seen the execution time increases considerably in most cases. The code size generally decreases. In most cases, the compiler is unable to find a schedule that can keep the processor busy and so the number of parallel iterations in the software pipelined loops reduces and this reduces the size of prologs and epilogs.

In certain benchmarks like DCT, the loops are already longer than six cycles and this has no effect on the performance. It is possible to optimize C code for performance under interrupt latency constraints by unrolling inner loops. We tried this with a few benchmarks and did see improvements on some benchmarks, with an increase of code size.

5.6 Linear Assembly and the Assembly Optimizer

We wrote basic linear assembly versions of several of the benchmarks and compared their performance to that of hand-optimized assembly. Linear assembly code is an assembly like language that can be written without considering parallel instructions, functional units, latencies,

and register usage. The assembly optimizer schedules the instructions for the parallel architecture to maximize throughput, taking into account the latencies and register assignment. The linear assembly performance comparisons are shown in Figures 7 and 8.

6 Application Benchmarking

One drawback of the reviewed papers was that performance results in the references are all based on execution times of DSP kernels and not whole applications. DSP kernels have high parallelism and are easy to map to VLIW architectures. The DSP system designer is more interested in the speed of the whole application. The execution times of the kernels may not truly indicative of the whole application that uses that kernel. We studied the relationship between the execution times of the kernel and application, by coding a close variant of the JPEG codec in C and then iteratively applying various optimizations to speed up the program.

6.1 Benchmark Application

We coded a compression scheme that is closely related to the JPEG specification. This codec accepts images in 16-bits per pixel format. The image is divided into 8 pixel by 8 pixel regions. The DCT transformation is applied to each region to extract the frequency components. The frequency components are encoded using a variable-length coding scheme that is similar to Huffman coding.

This benchmark is one of the applications at which the C6x processor is targeted. Hence performance of the processor on this benchmark is of interest to the DSP design community. Secondly, this benchmark will test the optimization capabilities of the TI compiler due to mix of code that is present. There are two highly parallelizable loops, the DCT and IDCT kernel. And

there are relatively frequently executed functions that are used for variable-length encoding. The statements in these functions have data-dependent control structures and they do not consist of loop structures. Hence techniques like software pipelining cannot be applied in an obvious manner.

In a typical system, the extraction of DCT coefficient and Huffman encoding would be done in the DSP and file I/O would be handled by the host processor. Hence we decided to use a simpler implementation free of extensive file format support. We settled on the JPEG-like codec presented in the book titled "The Data Compression Book," written by Mark Nelson.

The TMS320C6x provides 64 Kbytes of on-chip program memory and 64 Kbytes of on-chip data memory. So, we designed the data structures so that the data and code fit into the respective 64 Kbytes of on chip memory. The images used are 64 by 64 pixel 16-bit images.

6.2 Results

When we had the final ANSI C code that performed the functions we wanted, we set that as our baseline for comparison. The codec consists of a setup phase where image data is converted into the format required by the DCT routine, a DCT computation phase and coefficient encoding phase. The decompression routine consists of a coefficient decoding phase and an IDCT computation phase. After that various optimizations were done and we measured the speedups for each phase due to every optimization.

The results from the study are shown in figure 9. The X axis shows the optimizations that were tried. The Y axis shows the speedup over the baseline code that was achieved for each optimization for each component of the codec.

- The baseline code is the naïve code in which the DCT is implemented as two nested loops doing fixed-point multiplication with a naïve implementation of the encoding scheme.

- The second version was the same as baseline except that it used an optimized ANSI C version of the DCT and IDCT routines. This version still used the naïve coefficient encoder.

- The third version improved on the second version by optimizing the coefficient encoder and decoder.

- The fourth version used intrinsics instead of shift and masking to implement the coefficient encoder and decoder.

- The fifth version was compiled interrupt latency of 10 cycles and all previous optimizations.

- The sixth version used a hand-optimized assembly version of the DCT and IDCT routines obtained from TI's web site [9].

We can see from the graph that DCT/IDCT kernels show maximum speedups over the baseline case, going upto 23 times in the best case. But the encoding/decoding routines have the maximum impact on the overall execution time since they consume a large fraction of the total execution cycles. So we see that the total speedup in the best case is only about 6 times the baseline case. When we generate the code so that it is interruptible, we do not observe much loss in performance. This is because the software-pipelined loops in this program are already longer than 6 cycles.

7 Problems with the C Compiler

7.1.1 Loop Overhead

One of the reasons why the C compiler is not able to achieve the performance of hand-optimized assembly is loop overhead in software pipelined loops. Table 1 shows five optimized C versions of the FIR benchmark and corresponding execution times in clock cycles. (M and N are the number of outputs and the number of taps of the FIR. Usually M>>N)

Table 1: Clock cycle counts for various FIR implementations

Redundant Load Elimination	Outer Loop Unrolling Factor	Inner Loop Unrolling Factor	Intrinsics	Clock cycles	Overhead
	8	2	×	M(N+11)/2 + 22	11M/2+22
	4	2	×	M(N+9)/2 + 47	9M/2 + 47
	2	2	×	M(N+19)/2 + 19	19M/2+19
×	2	4		M(N+24)/2 + 24	12M+24
×	2	2		M(N+19)/2 + 19	19M/2 + 19

All the routines achieve a throughput of 2 MACs per cycle, but the second one has the least overhead. This overhead accounts for about a 25% increase in execution time. So, just optimizing the code to generate an optimal kernel is not sufficient. The compiler should use more information about the loop counter and try to get rid of this overhead. Hand-optimized assembly routines make use of such information (the loop count factor), and can completely remove the overhead in certain cases. Newer versions of the compiler support an enhanced `_nassert` intrinsic that can pass on information about the loop counter factor.

7.2 Compiler Heuristics

In certain benchmarks and in the JPEG codec, we observed significant counter-intuitive changes in the performance due to even minor changes in the program structure. The compilers heuristics behave in an unpredictable fashion especially when handling code that was not software-pipelined. This is most visible in the DCT benchmark. Re-compiling with an additional interrupt latency constraint actually improved the performance. Even small changes like re-ordering of independent instructions can affect the performance significantly (20%-30%). Optimization of the code then often becomes a trial and error based search for the best performance.

8 Summary

In this project we evaluated the performance of the TMS320C6x code-generation tools. We compared the performance of the C Compiler on various DSP kernels and measure the effect of various C optimizations. We also test the application tools on a real-world application, a JPEG coder-decoder. Our conclusions are listed below

- ANSI C code is approximately 2-10 times slower than hand-optimized assembly is. The TMS320C6x compiler is unable to really take advantage of the VLIW architecture unless the programmer optimizes his program just for the TMS320C6x.

- With the simpler benchmarks, C code optimization yields a speedup of about 2 times over ANSI C versions. In many of the simple benchmarks, this comes close to hand-optimized assembly.

- In the more complex benchmarks, like DCT and FFT, it is more difficult to make such optimizations, and we were not able to match similar performance.

With C optimization, code size generally increases, but we can offset this increase with epilog removal without a loss in performance.

- Linear assembly provides an effective way to utilize the resources of a VLIW processor but requires considerably more effort to program.

- In the JPEG application, significant speedups in the DCT and IDCT kernels had less of an impact on performance than smaller speedups in the Huffman coding. In the JPEG application, we also found that the DCT kernels and Huffman coder consumed most of the execution cycles, and the overhead was negligible (2%).

9 References

[1] P. Lapsley, J. Bier, A. Shoham, E. A. Lee, *DSP Processor Fundamentals*, Berkeley Design Technology, Inc., unpublished, Freemont, CA, 1996.

[2] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", *Proceedings of the SIGPLAN '88, Conference on Programming Language Design and Implementation*, Vol. 23, No. 7, (Atlanta, GA), pp. 318-328, June 22-24, 1988.

[3] P. Faraboschi, G. Desoli, J. Fisher, "The Latest Word in Digital and Media Processing", *IEEE Signal Processing Magazine*, Vol. 15, No. 1, pp. 59-85, March 1998.

[4] R. Simar, "Codevelopment of the TMS320C6x VelociTI Architecture and Compiler", *Proc. IEEE Conf. on Acoustics, Speech and Signal Processing*, Vol. 5, pp. 3145-3148, Seattle, WA, May 12-15, 1998.

[5] N. Seshan, "High VelociTI Processing", *IEEE Signal Processing Magazine*, Vol. 15, No. 2, pp. 86-101, March 1998.

[6] M. Levy, "C Compilers for DSPs Flex their Muscles", *EDN*, Vol. 42, No. 12, pp. 93-102, June 5, 1997.

[7] Longhborough Sound Images, PLC, *Evaluation of the Performance of the C62201 Processor and Compiler*, White Paper Version 1.1, 1997.

[8] *DSPS FEST '98 TMS320C6xxx DSP Design Mini-Workshop Student Guide*, Texas Instruments, July 1998.

[9] Texas Instruments Inc TMS320C62x Assembly Benchmarks, http://www.ti.com/sc/docs/products/dsp/c6000/62bench.htm

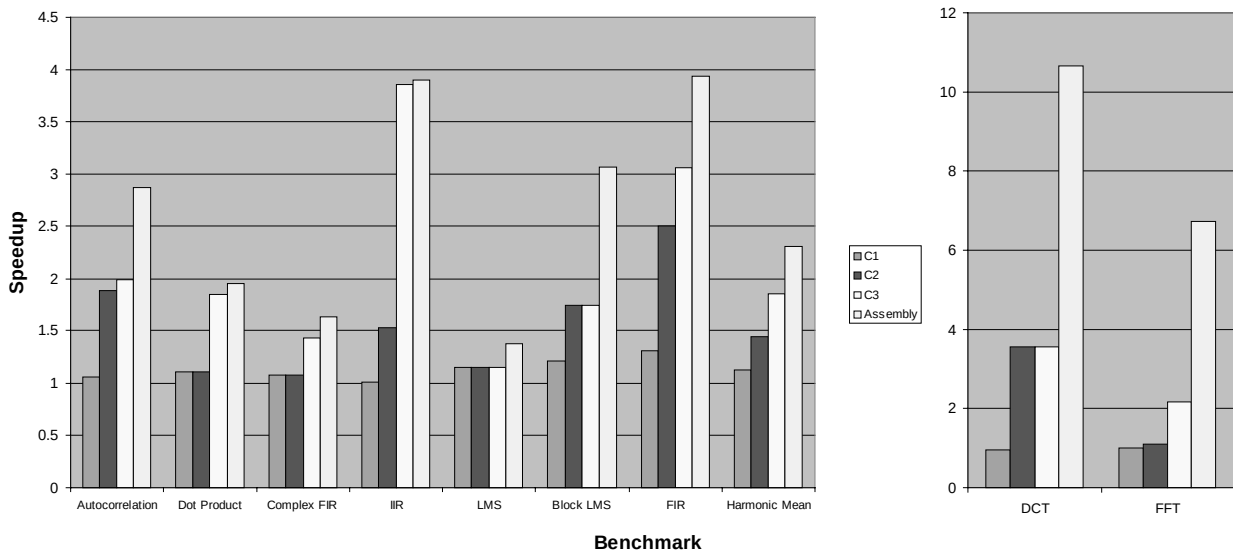

Figure 1: Speedup Over ANSI C

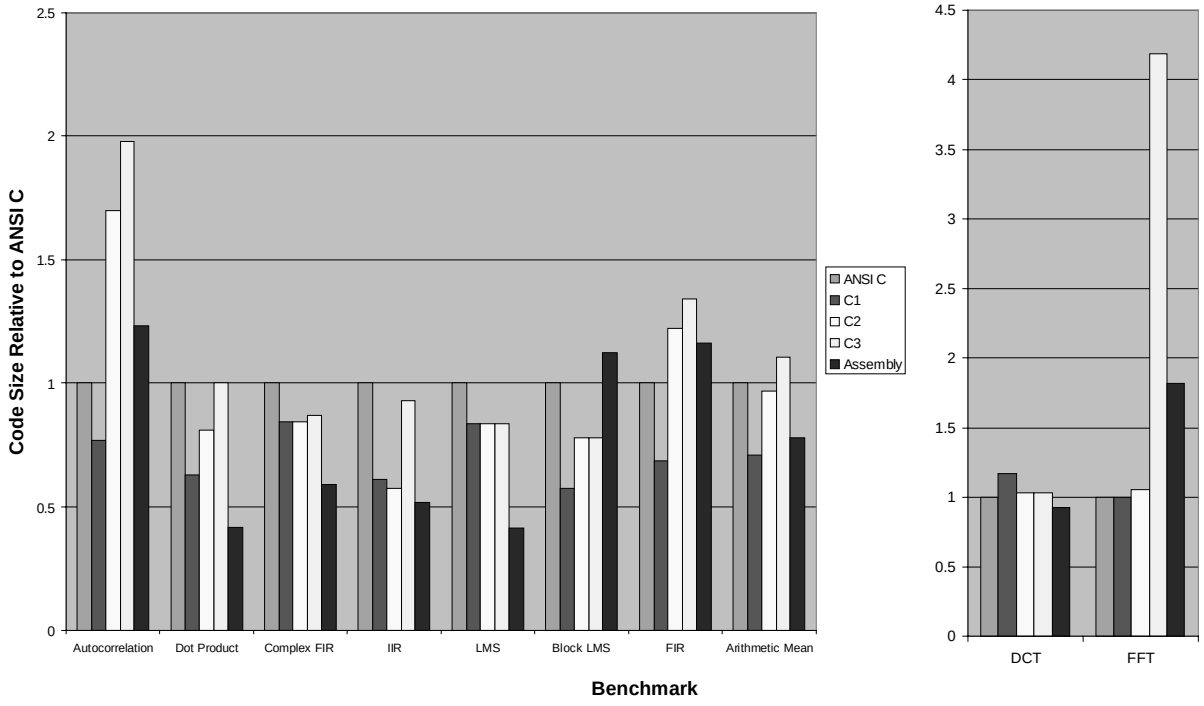

Figure 2: Code expansion (normalized to ANSI C)

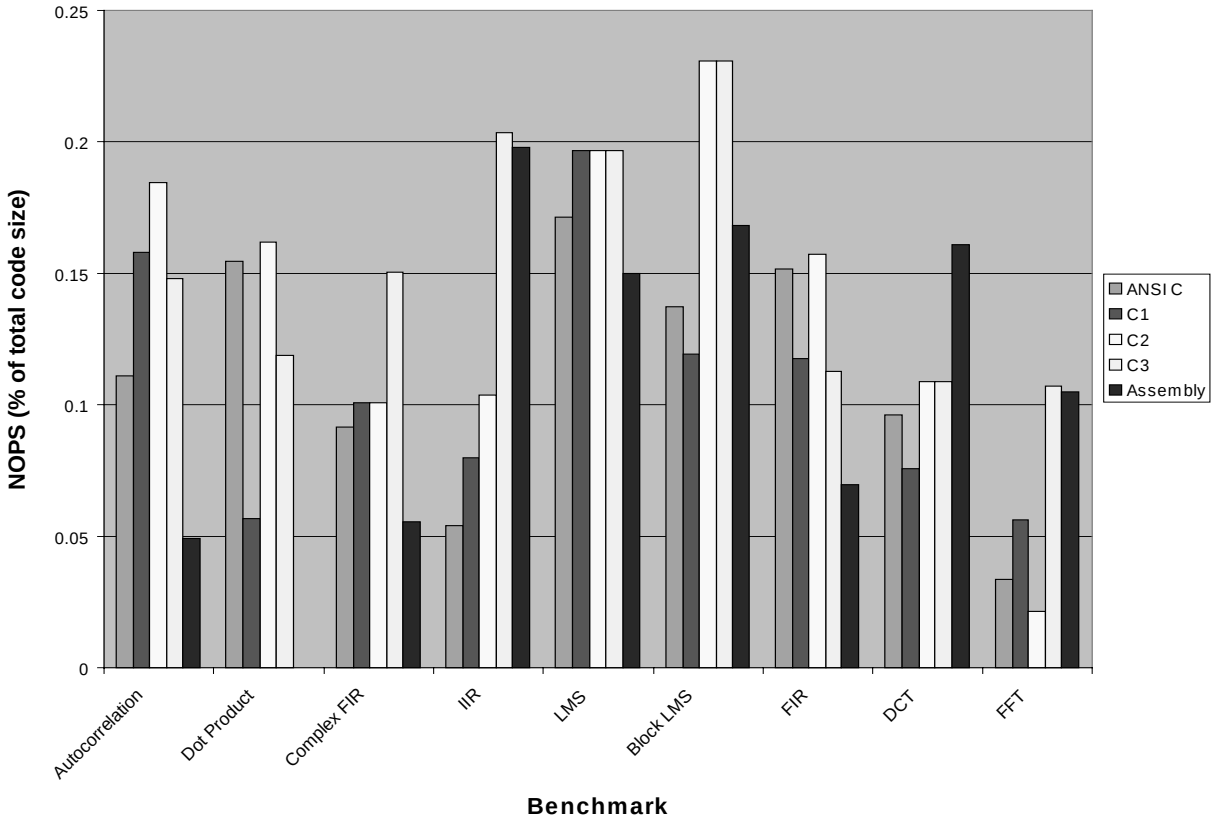

Figure 3: Unused program memory

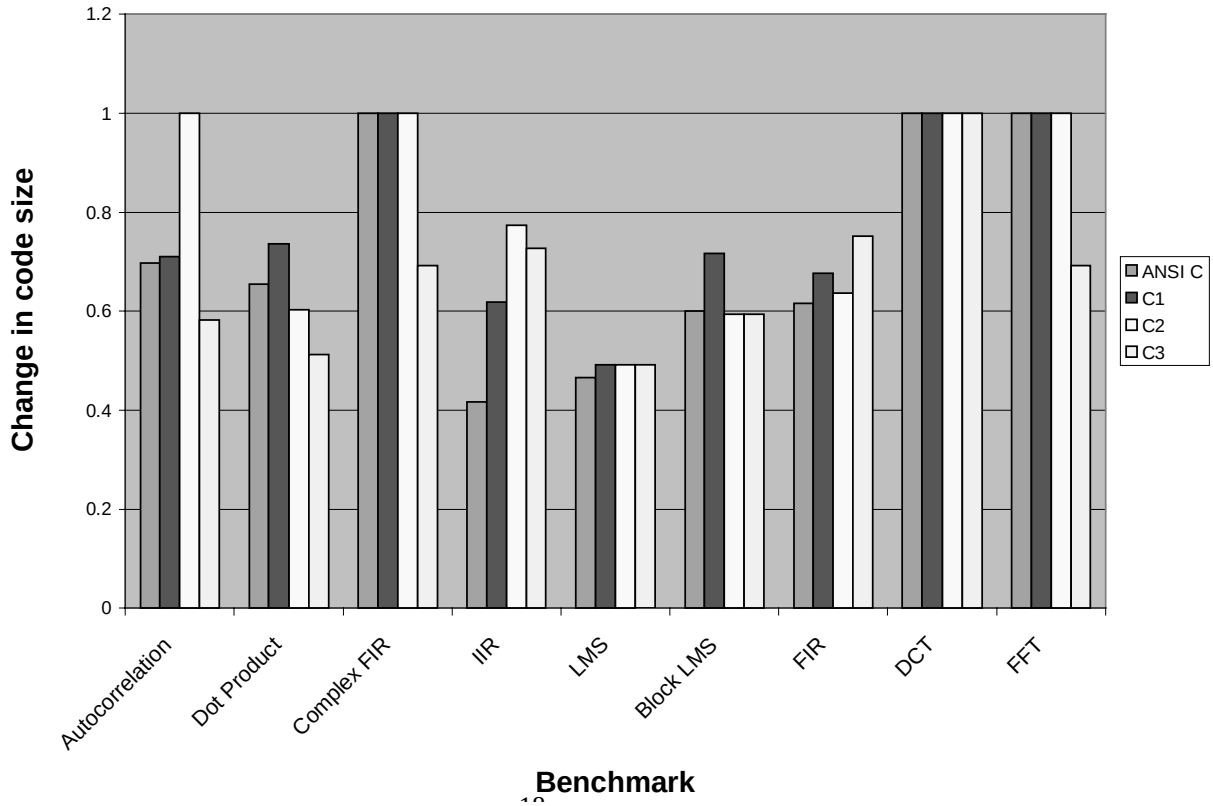

Figure 4: Effect of epilog removal on compiled C code

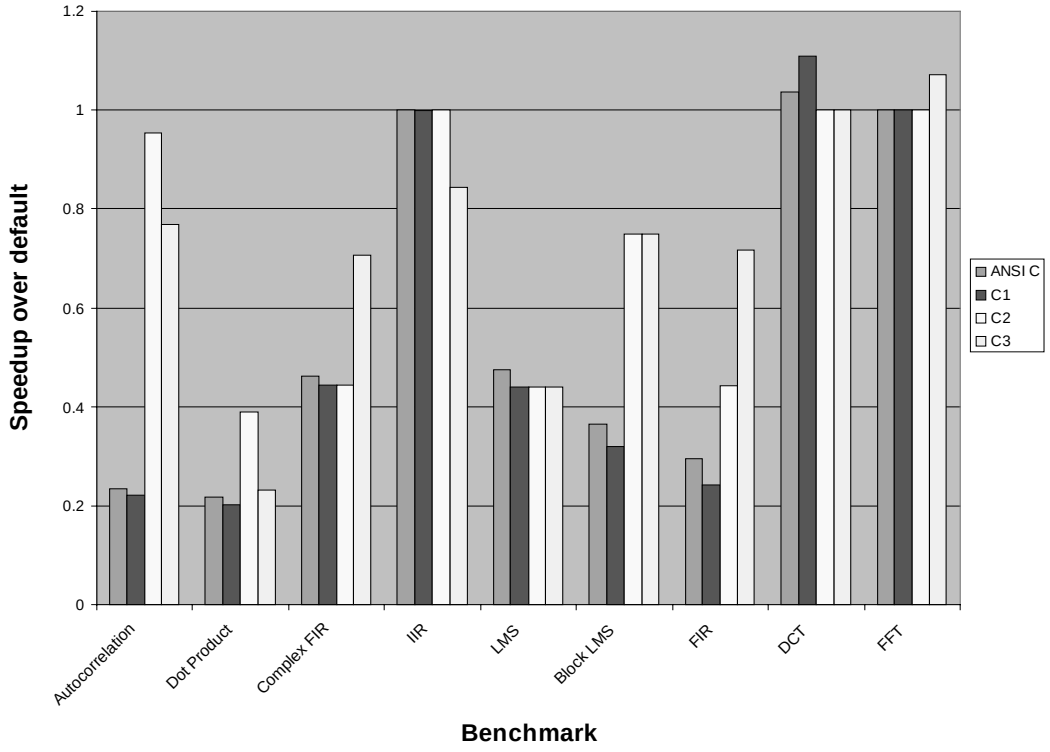

Figure 5: Change in execution time when interrupt threshold is set to ten

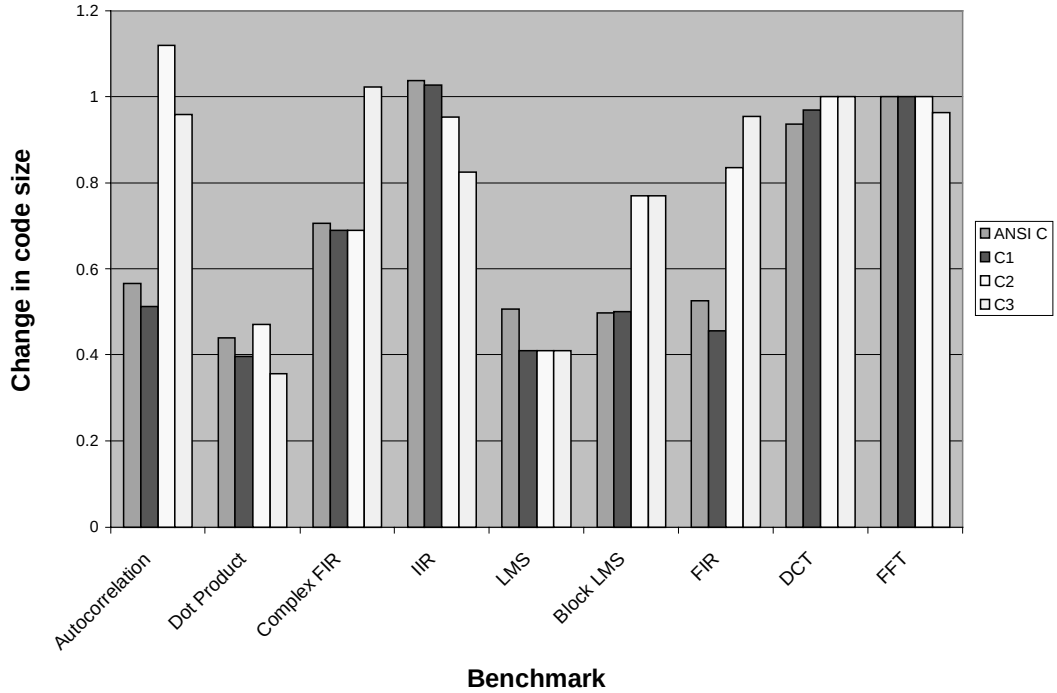

Figure 6: Change in code size when interrupt threshold is set to ten

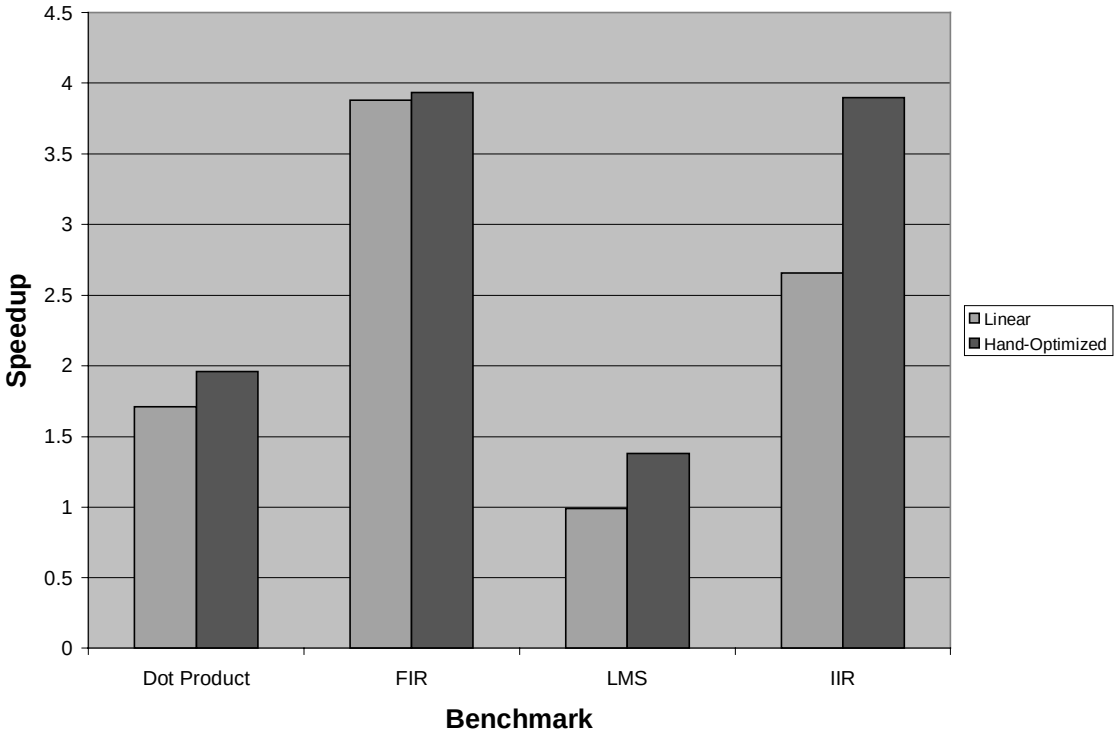

Figure 7: Speedup of assembly versions (over ANSI C) of several of the benchmarks

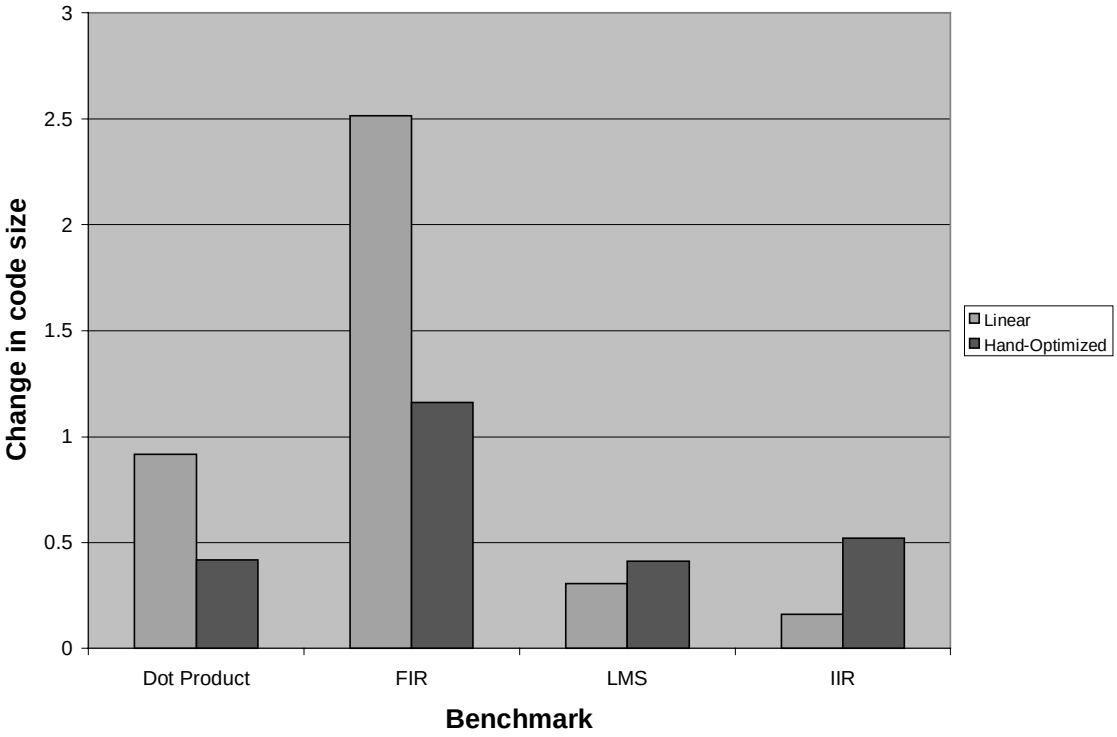

Figure 8: Change in code size (from ANSI C version) of assembly versions

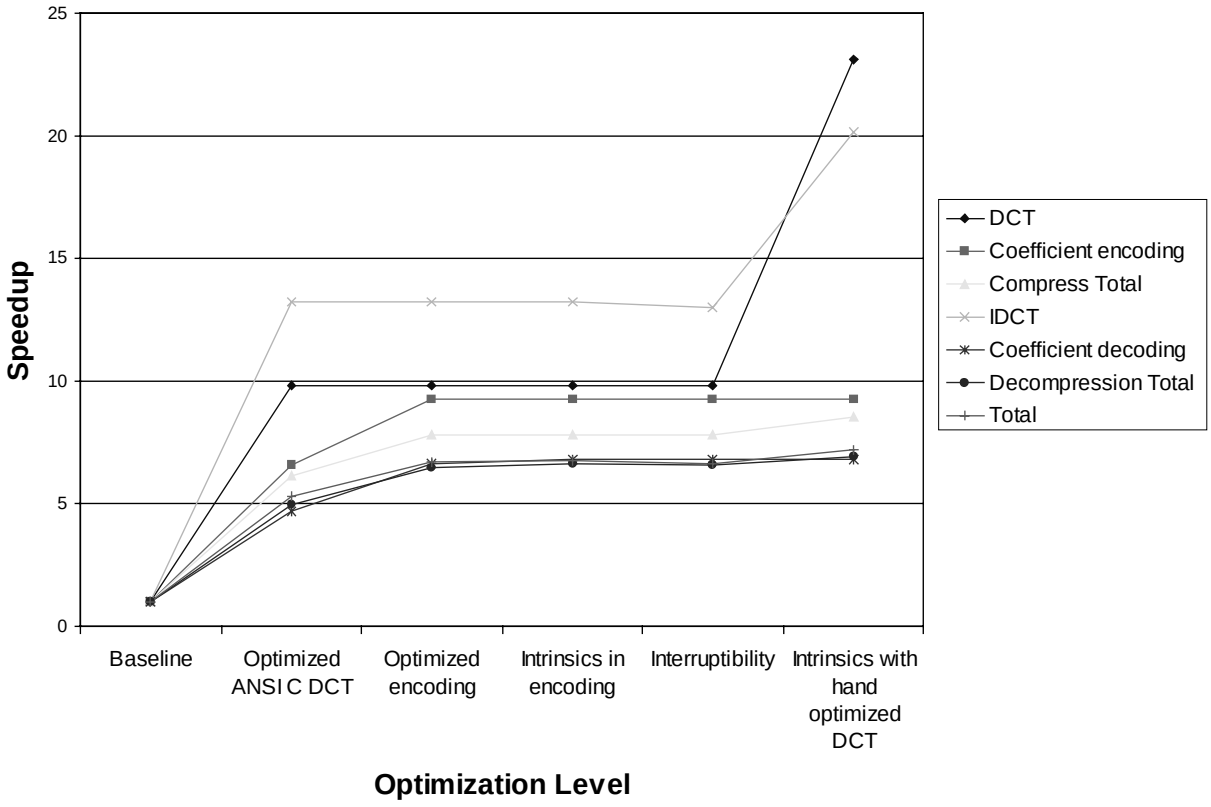

Figure 9: Effect of optimization stage on JEPG codec