BENCHMARKING CODE GENERATION METHODOLOGIES FOR PROGRAMMABLE DIGITAL SIGNAL PROCESSORS

Ashutosh K. Kulkarni, Aditya Dube

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712, USA

ABSTRACT

We evaluate rapid prototyping tools and compilers as code generation methodologies for programmable digital signal processors (DSPs). Code generated by compilers and rapid prototyping tools have been reported as significantly less efficient in memory usage and execution time versus assembly language code written by expert programmers. As the complexity of the system increases, however, the scale tips in favor of the automated code generation techniques. We quantify when this trade-off occurs by isolating the effects of (1) compiler inefficiencies and (2) automatic scheduling algorithms.

1. INTRODUCTION

For nearly two decades, programmable digital signal processors (DSPs) have performed computation and data intensive tasks in real-time with limited on-chip memory. In order to meet real-time constraints, programmers have traditionally written and optimized DSP assembly language code by hand. Sometimes, less time critical parts can be implemented in a higher-level language such as C. As systems become more complex, two alternatives become viable: (1) write the entire application in a higher-level language, and (2) use a rapid prototyping tool to generate software for the application.

We evaluate these two alternative code generation methodologies by isolating and measuring the factors that lead to implementation overhead. We perform the evaluation on the Motorola 56002 DSP using the GNU-based Motorola KCCA56 C compiler (Version 1.26; May 22, 1996) and the automated C and 56000 code generators in the UC Berkeley Ptolemy design environment (Version 0.7; June 13, 1997).

2. BACKGROUND

DSPs incorporate special architectural features such as bit-reversed addressing for FFT routines, modulo addressing for circular buffers, concurrent access to data and program memories for high throughput, and hardware looping for digital filters [1]. Many compilers have been unable to take full advantage of these features. For example, compilers for DSPs with two data memory banks show a bias of as much as 70% toward one data bank. Compilers have also been inefficient in exploiting parallelism in memory access and address calculations. Hence, code generated by compilers often suffers from large run time overhead.

Ptolemy, a rapid prototyping tool, automatically generates code in C (CGC domain) and in the Motorola 56000 assembly language (CG56 domain). These domains obey Synchronous Dataflow (SDF) semantics [5] which facilitates static scheduling and code optimization. Code generation occurs in two phases: (1) scheduling, in which the functional operations used in specifying the system are scheduled, and (2) synthesis, in which the code segments for the functional operations are stitched together for execution on the target processor [2]. Architecture-specific features of the target can be leveraged during the synthesis phase.

3. BENCHMARKING METHODOLOGIES

We measured the cost of four implementation techniques to evaluate the relative merits of code generation methodologies. Each of the benchmark programs were coded by

- programming manually in 56002 Assembly Language,
- compiling hand-written C code using the Motorola KCCA56 compiler,
- using Ptolemy's CG56 domain for generating 56002 assembly language code, and
- using the KCCA56 compiler on a C implementation generated by Ptolemy's CGC Domain.

We isolated the effects of (1) compiler inefficiencies and (2) automatic scheduling algorithms. Comparing assembly language and C implementations exposes compiler deficiencies, whereas comparing hand-written code with Ptolemy generated code emphasizes differences in manual vs. automated scheduling algorithms. For example, by comparing the results for the same application given by the Ptolemy CG56 domain vs. the Ptolemy CGC domain plus the KCCA56 C compiler with the same scheduler, we isolate the effects of the KCCA56 C compiler.

3.1 Benchmarking with Kernels

We evaluated three kernels as shown in Tables 1-3: (1) 5-tap IIR filter, (2) 256-point complex FFT, and (3) Goertzel's DFT Algorithm. We observe a substantial increase in memory usage with compilers because of the additional layer of abstraction between the software developer and the application. Data memory overhead varied from 0% to 66% with an average value near 27%. The average program memory overhead of 41% increased with the complexity of the implemented kernel. Programs generated by hand or Ptolemy do not differ greatly in size or execution time. Also the code generated by Ptolemy does not show a bias towards one data memory bank versus the other. This is to be expected as the code for kernels in Ptolemy is drawn from optimized libraries which do not allow such a bias to exist. Some overhead is inevitable with compilers. When we use kernels for benchmarking, Ptolemy fares much better because of the optimized code from its libraries. The power of Ptolemy's optimizing scheduler is not utilized as these kernels consist of only a few functions.

	Program Memory	X Memory	Y Memory	Exec. Time
Hand Coding in Assembly	43	7	8	517
Ptolemy CG56	49	7	8	561
Hand Coding in C	59	11	11	1127
Ptolemy CGC	57	11	12	963

Table 1. Memory usage and Execution times for IIR Filter kernel

	Program Memory	X Memory	Y Memory	Exec. Time
Hand Coding in Assembly	130	67	60	29172
Ptolemy CG56	136	67	65	25661
Hand Coding in C	178	69	77	30927
Ptolemy CGC	193	68	69	32631

Table 2. Memory usage and Execution times for 256 point complex FFT kernel

	Program Memory	X Memory	Y Memory	Exec. Time
Hand Coding in Assembly	67	41	3	17341
Ptolemy CG56	67	37	3	17283
Hand Coding in C	111	39	5	20123
Ptolemy CGC	102	43	3	19487

Table 3. Memory usage and Execution times for Goertzel's DFT kernel

3.2 Benchmarking with Applications

Using stand-alone applications is more realistic for benchmarking methodologies than using kernels. We have chosen three applications to represent the complexity and style of typical DSP applications. These were demonstrations in Ptolemy which follow Synchronous Dataflow semantics:

- IIR DEMO (Two ways of implementing an IIR filter) - Figure 1 and Table 4.
- CD-DAT converter (Multirate application, 44.1kHz (48kHz) - Figure 2 and Table 5.
- Dual-Tone Multiple Frequency (DTMF) touchtone Codec (Communications application) - Figure 3 and Table 6.

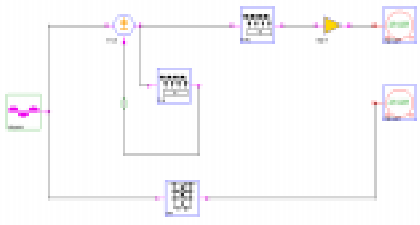

Figure 1. Ptolemy schematic for IIR Demo

We found that compilers cause an explosion of program memory size between 40% and 119%. The low figure of 40%, which is for the CD-DAT stand-alone C implementation is quite deceptive. In C code, the FIR filter implements polyphase resampling. In the CG56 domain, the FIR filter can

perform *either* interpolation *or* decimation, so an additional Downsampling star is needed.

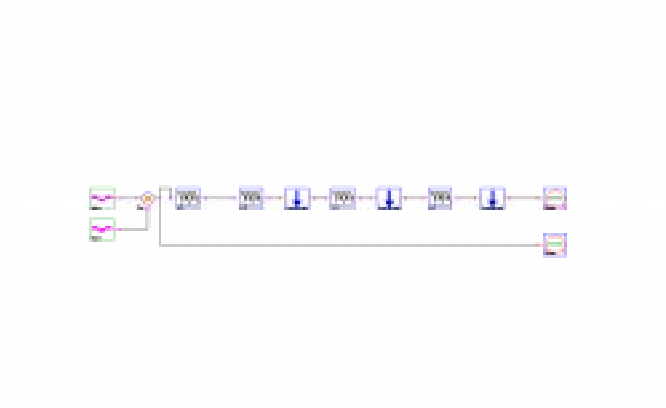

Figure 2. Ptolemy schematic for CD-to-DAT Converter

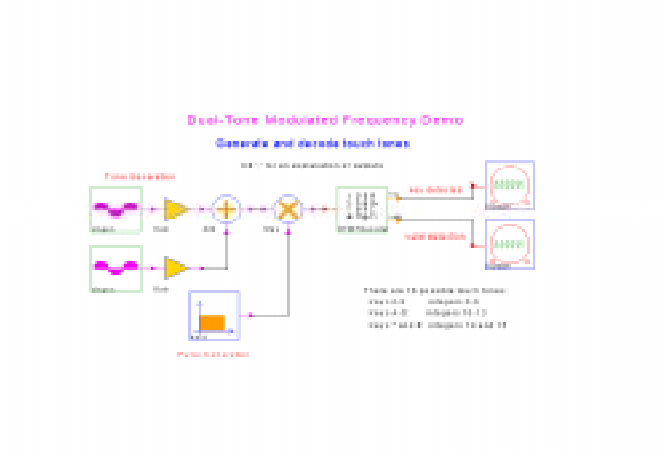

Figure 3. Ptolemy schematic for DTMF Codec

	Program Memory	X Memory	Y Memory	Exec. Time
Hand Coding in Assembly	86	20	30	41946
Ptolemy CG56	88	20	33	42031
Hand Coding in C	143	21	31	67132
Ptolemy CGC	152	21	42	69246

Table 4. IIR Demo - Memory usage and Execution times

We found that the KCCA compiler efficiently utilizes 56002 addressing modes, especially the bit-reversed addressing mode (complex FFT in the IIR Demo). For IIR filters, we observed a repeated exploitation of hardware looping. Our experiments agreed with [8] that the better DSP compilers handle data and variables efficiently with the worst case increase in data memory size being about 9%.

	Program Memory	X Memory	Y Memory	Exec. Time
Ptolemy CG56	413	456	283	295069
Ptolemy CGC	586	468	280	381076
Hand Coding in C	687	398	324	463004

Table 5. CD-to-DAT Converter Memory usage and Execution times

	Program Memory	X Memory	Y Memory	Exec. Time
Ptolemy CG56	976	3486	46	2195195
Ptolemy CGC	2146	3824	58	5431957
Hand Coding in C	40753	3392	54	***

*** The large size of the code generated made it impossible to execute the program on our system because of program memory limitations.

Table 6. DTMF Codec Memory usage and Execution times

Ptolemy includes libraries of highly optimized code modules which its code generation domains 'stitch together' according to a schedule. The scheduler possesses inherent knowledge of the communications among the subsystems allowing Ptolemy to better utilize scarce on-chip memory. The distribution of data memory is more equitable in the Ptolemy CG56 implementations of the IIR DEMO and CD-to-DAT programs than in the stand-alone C implementations. With increasing complexity, scheduling assumes greater importance. An intelligent scheduling heuristic (such as the one in Ptolemy that jointly minimizes program memory and data memory size) is likely find a far better schedule and code in complex cases. This is why the CGC implementation in the CD-to-DAT program fares better than the stand-alone C code.

Ptolemy-generated 56000 assembly language programs outperform compiled hand-coded C implementations in program memory usage and execution time, and are comparable in data memory usage. The gap widens as

complexity increases. At some point between the complexity of the IIR filtering and CD-DAT converter applications, Ptolemy-generated C programs begin to outperform hand-coded C implementations in the same way. The increase of complexity from a CD-DAT converter to a DTMF codec causes an increase by an order of magnitude of the efficiency of Ptolemy-generated C programs over the hand-coded C implementations. Although not shown, we found that for the IIR filtering demonstrations that Ptolemy-generated assembly language programs were only 2% worse in performance vs. hand-coded assembly implementations.

4. CONCLUSIONS

As far as which code generation methodology to use to create the most efficient implementations in a DSP assembly language, we draw the following conclusions:

- When the choice is between writing C code manually for compilation and using a synthesis tool to generate assembly language directly, the synthesis tool should be chosen.
- A key use of a C compiler is to complement a tool that synthesizes assembly language programs by generating efficient implementation of kernels in assembly language when they do not exist.
- When the choice is between writing assembly language code manually and using a synthesis tool to generate assembly language, the synthesis tool should be chosen for applications of complexity slightly greater than a DTMF codec.

Placing the responsibility of finding a schedule for a complex, multirate application on the programmer can often lead to extremely inefficient implementations. Intelligent schedulers provided in a tool like Ptolemy ensure that the code produced will be near optimal in program and data memory size. Based on our results, an ideal environment for software development on programmable DSPs would use a rapid prototyping tool at the system level (e.g. a dataflow graph in Ptolemy) and a highly optimized compiler (e.g., the KCCA56 compiler) at a finer level of granularity to extend the libraries provided by the tool as the better compilers for DSPs add very little overhead. When the overhead is not acceptable, hand coding would be used to add time critical functions to the libraries of the tool.

5. REFERENCES

[1] M. A. R. Saghir, P. Chow and C. G. Lee, "Application-Driven Design of DSP Architectures and Compilers," *Proc. of IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, vol. 2, pp. 437-440, Apr. 1994.

[2] J. L Pino, "Software Synthesis for Single-Processor DSP Systems using Ptolemy," *Master's Report*, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA 94720, 1994.

[3] S. S. Bhattacharya and E. A. Lee, "Memory management for Dataflow Programming of Multirate Signal Processing Algorithms," *IEEE Trans. on Signal Processing*, vol, 42, no. 5, pp. 1190-1201, May 1994.

[4] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, vol. 12, no. 8, pp. 971-989, Jan. 1987.

[5] S. S. Bhattacharya, P. K. Murthy and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Press, 1996.

[6] J. L. Pino, S. Ha, E. A. Lee and J. T. Buck, "Software Synthesis for DSP using Ptolemy," *Journal on VLSI Signal Processing*, vol. 9, no. 1, pp. 7-21, Jan. 1995.

[7] S. S. Bhattacharya, J. T. Buck, S. Ha and E. A. Lee, "Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms," *IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications*, vol. 42, no. 3, pp. 138-150, Mar. 1995

[8] P. Lapsley, J. Bier, A. Shoham and E. A. Lee, *DSP Processor Fundamentals - Architectures and Features*, Berkeley Design Technology, Inc., 1996.