Code Generation for TMS320C6x in Ptolemy

Sresth Kumar, Vikram Sardesai and Hamid Rahim Sheikh
EE382C-9 Embedded Software Systems
Spring 2000

Abstract

Most Electronic Design Automation (EDA) tool vendors have recognized the importance of software synthesis for programmable devices. Ptolemy is an object-oriented platform developed by University of California, Berkeley, which provides a block-diagram mechanism for representing systems described by one or more models of computation. It allows simulation of systems as well as synthesis of software from a block diagram for a variety of target languages: low-level assembly as well as high-level languages. In this project, we implemented code generation for the Texas Instrument's TMS320C6x family of processors in Ptolemy from a system description with Synchronous Dataflow (SDF) semantics, by employing optimized kernels for common Digital Signal Processing (DSP) operations. We evaluated the performance of our scheme for various configurations of these blocks.

1. Introduction

Ptolemy is an object-oriented framework for simulating, prototyping and synthesizing heterogeneous systems. Ptolemy's Code Generation environment consists of a number of *domains* pertaining to different models of computation and target processors (or languages) and architectures. Systems are described as block diagrams in the desired code generation domain. Executing the system generates code for that processor targeting a particular platform, e.g. single processor, multiple processors or desktop simulation.

In this project, we implemented code generation for the Texas Instruments' (TI) TMS320C6x (C6x) processors in the Ptolemy's C Code Generation (CGC) domain. Our approach results in significant execution speed improvement over existing CGC blocks.

2. Synchronous Dataflow (SDF)

SDF is a dataflow model in which the number of tokens produced and consumed by each actor in a graph upon execution is known beforehand and is fixed (and finite) throughout the execution of the graph. An actor cannot be executed until all of its inputs have the required number of data tokens [1]. These features allow bounded memory execution and static scheduling of valid SDF graphs. Multiple sampling rates can be conveniently handled in a system description adhering to the SDF semantics.

Optimal scheduling of most practical SDF graphs onto a single processor can be performed in polynomial time. Heuristics exist that can achieve or approach the lower bound for program and data memory requirements. Computational resource requirements are known at compile time and are static throughout the execution of the graph [2]. In this project we focus solely on the SDF model of computation.

3. TMS320C6x VLIW RISC DSP

Very Long Instruction Word (VLIW) *Reduced Instruction Set Computing* (RISC) architecture is very popular in the DSP world because of its ability to take advantage of the *Instruction Level Parallelism* (ILP) that is available in typical DSP programs while simplifying the control logic. While compilers perform poorly for traditional DSPs, those for VLIW RISC processors are more efficient because of trace scheduling and software pipelining [3].

The TMS320C6x family is a high-end multimedia VLIW RISC processor family whose architecture and C compiler were co-designed. The C6x and its C Compiler were designed to ease the task of programming complex applications. Programming the C6x in assembly is very cumbersome for managing these complex tasks, esp. because of its eight parallel execution units and a pipeline of 11-16 stages. TI provides optimized assembly code for common DSP structures such as Finite Impulse Response (FIR) filters and Infinite Impulse Response (IIR) filters.

4. Code Generation for C6x

The functionality of code generation actors lies in their *codeblocks* which contain generic C code in the case of CGC domain, which would appear in the final synthesized code. However, this code would not be optimized for any particular processor. Synthesizing efficient code requires these codeblocks to be optimized for a target architecture and compiler [4].

We address this problem by embedding function calls to a library of optimized C-callable assembly kernels provided by Texas Instruments (TI) for common DSP operations. This library can be upgraded without any change in the system or applications. This approach is particularly advantageous for processors that are very difficult and tedious to program directly in assembly.

Ideally, each supported processor should have a separate code generation domain so that Ptolemy can support implementation-independent design and handle constraints such as

processor specific data types. However, we confined our work to adding a new target and stars to the CGC domain.

5. Implementation

We used the CGC domain as our basic framework and built a C-callable library of functions optimized for the C6x processors. When the generated C-code is linked across this library, optimized executable code is synthesized. Presently, the library contains functions for single rate FIR & IIR filters. We implemented multirate FIR filter by using a combination of single rate FIR filters. TI's assembly kernels have certain restrictions, which are also inherited by our stars.

Texas Instruments' FIR kernel had to be fixed for two bugs. In the first case, the address mode register was being set up in parallel with an instruction that required the new value of the same. In the second case, an instruction resetting a circular buffer pointer was performing linear instead of modulo subtraction.

Using the basic FIR filter we also implemented a decimator, interpolator and a sampling rate converter. This was done in order to meet our objective of implementing a 44100:48000 sampling rate converter which is used for conversion from Compact Disc (CD) to Digital Audio Tape (DAT) format. We compared a C6x version with an equivalent implementation that uses existing CGC stars.

5.1 Polyphase implementation

The decimator, interpolator and the sampling rate converter were implemented using efficient polyphase representation. The Polyphase implementation of a decimator (interpolator) with decimation factor (interpolation factor) *D* (*U*) is *D* (*U*) times more efficient than the direct form implementation. A *U/D* sampling rate converter implemented in polyphase form is *U*D*

times more efficient than the direct form implementation. The CGC multirate FIR star also employs polyphase representation for efficient implementation [5].

Figure 9 illustrates polyphase implementation for an interpolator and figure 11 illustrates the same for a decimator. Figure 10 illustrates how to commute a delay through an upsampler and a downsampler. A single delay (z^{-1}) can be decomposed into a *U*u* advance and a *D*d* delay subject to *U*u + D*d = -1* being satisfied for integer *u* and *d*. The upsampler followed by an advance of *U*u* is equivalent to an advance of *u* followed by the upsampler. Similarly, a delay of *D*d* followed by the downsampler is equivalent to the downsampler followed by a delay of *d*. Then, if *U* and *D* are co-prime, the upsampler and downsampler can commute. This way an *L*-tap FIR filter between an upsampler and a downsampler (figure 9a) can be decomposed into *U*D* single rate FIR filters, each of length *L / (U*D)* samples.

5.2 Target

We made a *target* class "c6xtarget" and instantiated an object named C6X. This target is visible in the CGC's target menu. Simulator specific instructions can be inserted in the C code before or after each actor firing. We could not test it completely since we did not have the C6x compiler and simulator for Unix. So the comparisons have been made on Code Composer Studio running on Windows NT.

6. Results

We compared the performance of code generated by our C6x stars with that of code generated by CGC stars in terms of cycle count, code size and data size. All comparisons are with level-3 optimizations. Since Ptolemy does not support *short* as a data type, comparisons were made for floating-point stars only. Since CGC generates code with *double* data type, we manually replaced all instances of *double* with *float*. However, we could not get the CGC IIR

universe to run properly with these replacements because the –O3 compile option skips the SDF loop altogether. Hence, for IIR filters, the comparisons are for the *double* data type for CGC.

6.1 FIR Filters

Figures 1, 2 and 4 show the cycle counts, code sizes and data sizes respectively for FIR filters versus filter length for CGC and C6x FIR stars. In terms of cycle counts, an improvement of 10 to 40 times is observed (figure 3). An asymptotic improvement in performance of 62 times is observed. This substantial improvement in speed is expected since the C6x compiler cannot understand and exploit the global computational structures of FIR filters from the C implementation. Therefore it cannot allocate DSP specific features such as circular buffers for an efficient implementation.

6.2 IIR Filters

Figures 5, 6 and 8 show the similar performance graphs for IIR filters. A speed improvement of 4 to18 times is observed (figure 7). The CGC IIR stars uses a 5-multiply IIR structure whereas the C6x IIR star uses a 4-multiply structure by combining the scale factors from all biquad sections into one. A large increase in code size for the CGC star is also observed because the scheduler generates code for each instance of a biquad star, whereas for the C6x star one instance handles all of the biquad sections.

6.3 CD to DAT sample rate conversion universe

Our project goal was to demonstrate the working of C6x stars by integrating them into a CD to DAT format conversion demonstration. The following table summarizes the comparison between CGC and C6x CD to DAT universes. Running the CGC CD-to-DAT universe with the *float* type and then with the *double* type, we inferred that the former runs approximately 1.4 times faster.

CD to DAT demo	CGC float/double	C6x	Remarks
Cycle count per sample	7290/10020	1400	C6x is 5.2x faster
Code size	15930/--	68030	4.3x
Data size	327860/--	334760	Almost same

One restriction that the C6x FIR star inherits from the TI's routine is that its length must be a multiple of 4. Most of the polyphase component filters in the demonstration require only a single tap but the C6x star does four times more computation. For example, for the 5:7 converter, a 140-tap filter would run as fast as the 32-tap filter used in the demonstration. Thus, in a more computationally intensive system, the C6x stars are expected to perform even better than their CGC counterparts. As an illustration, the upsampler (2:1) in the demonstration has 173 taps and its C6x version runs 34 times faster.

7. Conclusion

We have demonstrated that a library-based approach to code synthesis for complex processors is superior to generic C code synthesis, esp. for computationally intensive DSP algorithms. This approach also allows upgradability and scalability. We realized that compilers still lack the ability to exploit highly regular computational structures like FIR or IIR filters.

8. References

[1] E. A. Lee, D. G. Messerschmitt, "Synchronous Data Flow", *Proc. of the IEEE*, vol. 75, no. 9, pp. 1235-1245, Sept. 1987.

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

[3] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Ccompiler" *IEEE Trans. on Computers*, pp. 967 – 979, 1991.

[4] Jose Luis Pino, Soonhoi Ha, Edward A. Lee, Joseph T. Buck "Software Synthesis for DSP Using Ptolemy" *Journal of VLSI Signal Processing*, vol. 9, pp. 7-21, 1995.

[5] Sanjit K. Mitra, *Digital Signal Processing - A Computer Based Approach*, The McGraw-Hill Companies, Inc., 1998.

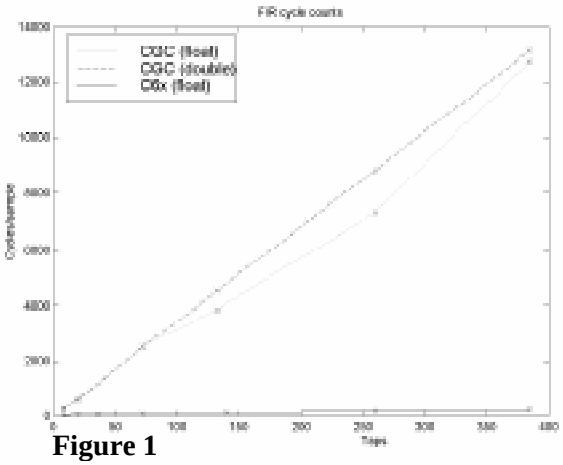

Figure 1

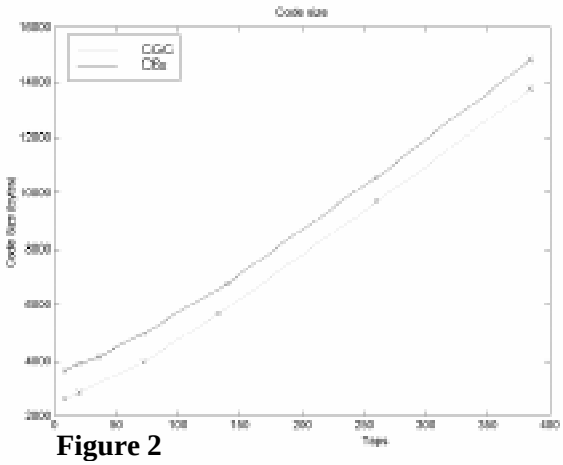

Figure 2

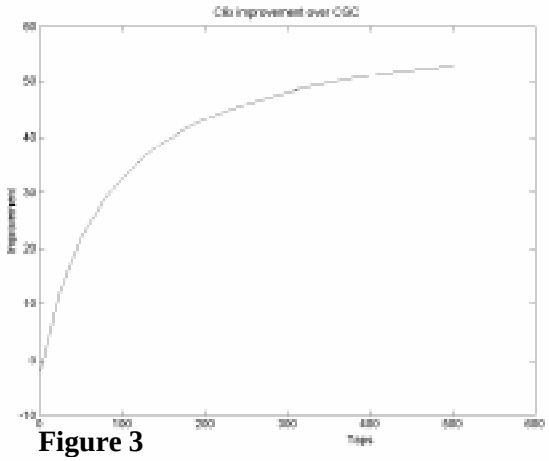

Figure 3

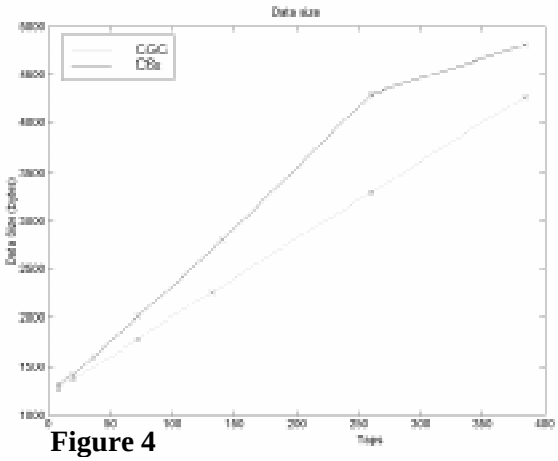

Figure 4

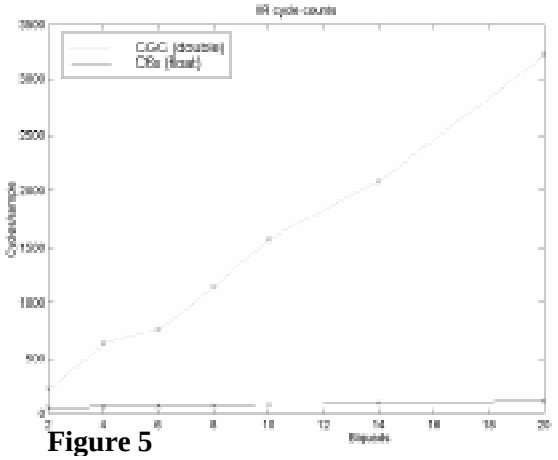

Figure 5

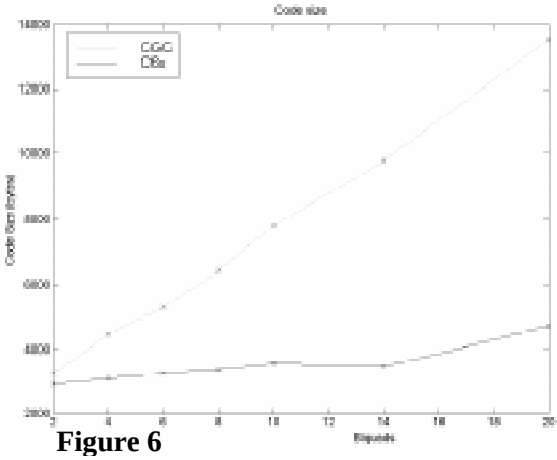

Figure 6

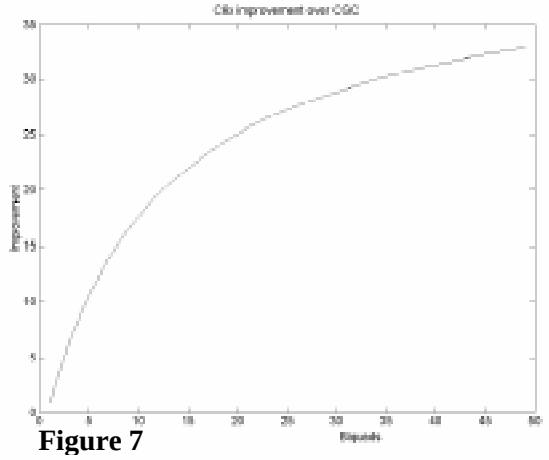

Figure 7

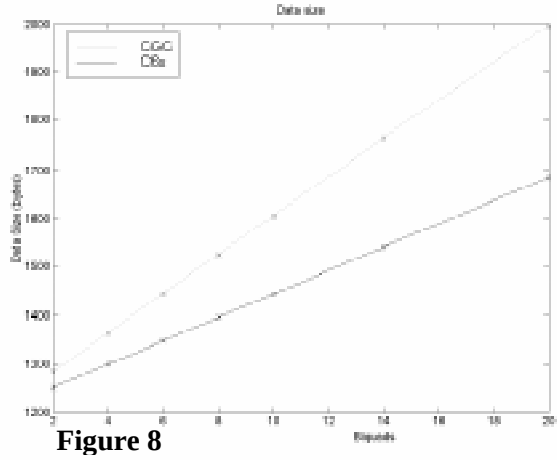

Figure 8

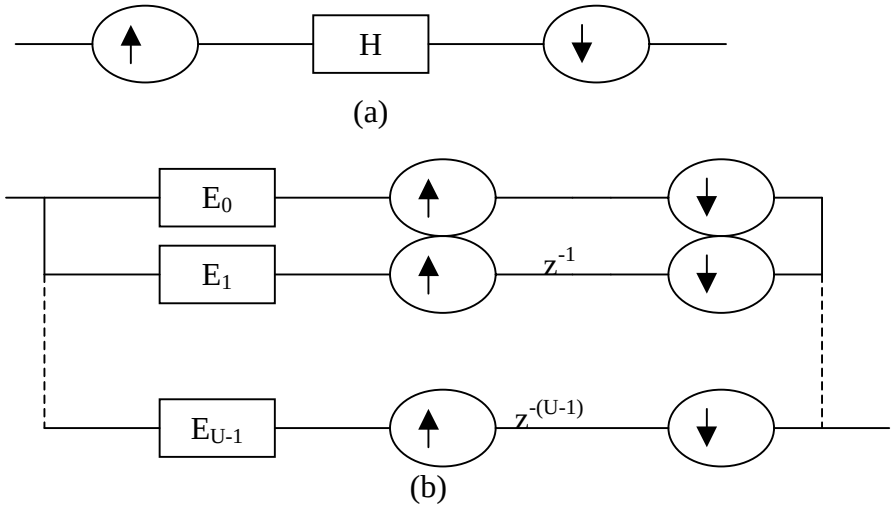

Figure 9: Multirate FIR filter cascaded with upsampler and downsampler.
(a) Direct form
(b) Polyphase expansion for the upsampling operator.

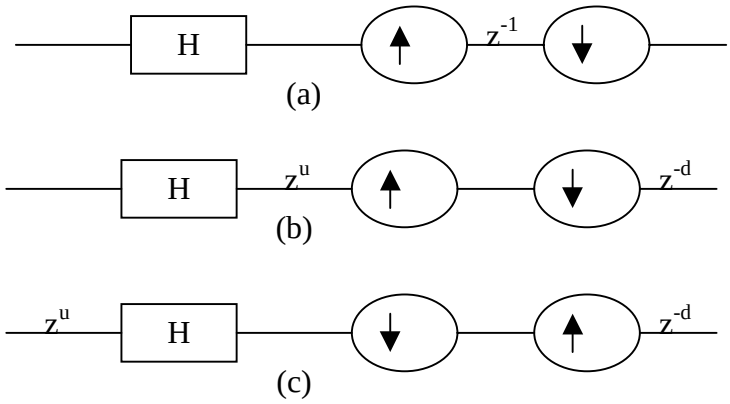

Figure 10: Commuting upsampler and downsampler through delay.
(a) The problem
(b) Taking delay out such that U*u - D*d = -1
(c) Commuting upsampler and downsampler.

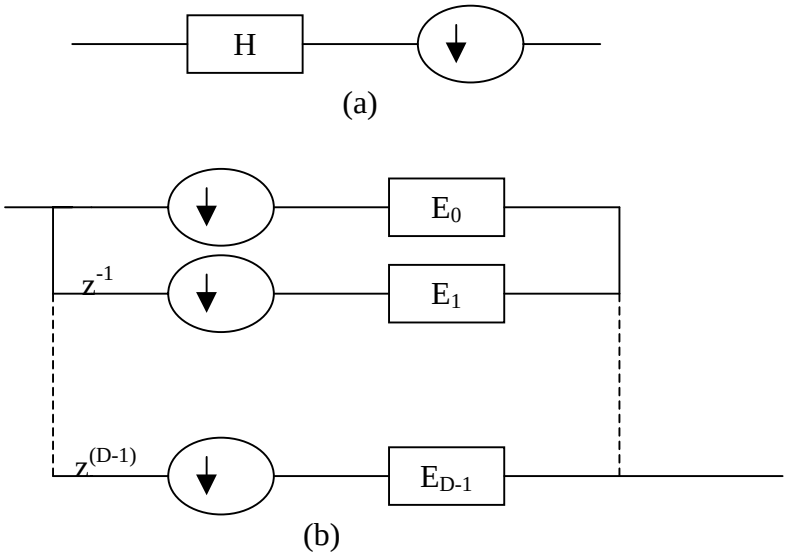

Figure 11: FIR filter followed by a downsampler.
(a) Direct form
(b) polyphase form.