Literature Survey
On

CGC6000

Ptolemy Code Generation Domain for TMS320C6x

Sresth Kumar
Vikram Sudhir Sardesai
Hamid Rahim Sheikh

EE382C-9
Embedded Software Systems
Prof. Brian L. Evans
Department of Electrical & Computer Engineering
The University of Texas at Austin.

March 2000.

Abstract

Ptolemy is an environment that 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 the block diagram for a variety of target languages: low-level assembly as well as high-level languages. A broad class of Signal Processing & Communication Applications can be described using the *Synchronous Dataflow* (SDF) model of computation. In this project, we intend to write a 'Code Generation Domain' for Ptolemy that will generate code for Texas Instrument's TMS320C6000 family of processors from a system represented as an SDF Graph.

1. Introduction

Ptolemy is an object-oriented framework for simulation, prototyping and synthesis of heterogeneous systems. It provides a mechanism for representing a system described by one or more *Models of Computation.* In this project, we are interested only in Ptolemy Code Generation environment.

Ptolemy Code Generation environment is modular and extensible because of its object-oriented design. It consists of a number of *domains* pertaining to different target processors (or languages) and architectures, e.g. CGC (Code Generation in C), CG56 (Motorola's DSP56k), C50 (TI's TMS320C50) etc. 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 target, multiprocessor target, and simulator target etc.

Texas Instrument's TMS320C6x is a VLIW RISC processor that is becoming increasingly popular because of its computational power and ability to handle demanding Signal Processing and Multimedia applications like DVD, MPEG, and Digital TV etc. Writing optimized code for the C6x manually is cumbersome and time consuming. In the modern era of constrained time-to-market schedules, efficient Automatic Code Generation mechanisms are very desirable.

In this project, we intend to write a Code Generation Domain for the C6x processor. Our domain would generate efficient C6x code from an SDF description of a system. The generated code would run on a C6x evaluation board and perform better than the code generated by the CGC.

2. Synchronous Dataflow (SDF)

In the Dataflow model of computation, a system is represented as a *graph*. Operations on data are represented as nodes (blocks or *actors*) of the graph and the arcs connecting the nodes are the data (or signal) paths. Communication between nodes is by means of data samples (or *tokens*) that travel along arcs. Dataflow is a natural paradigm for describing a large class of DSP systems. It exposes the parallelism that exists in most DSP algorithms, thus allowing designers to map the algorithm onto multiple processors if such an implementation is desired.

SDF is a dataflow model in which the number of tokens produced and consumed by each actor in the graph upon execution is known beforehand and is fixed (and finite) throughout the execution of the graph. An actor cannot be executed until all its inputs have the required number of data tokens. At each invocation (or *firing*) of the actor, a fixed number of tokens is consumed from the input arcs and a fixed number of tokens is produced on the output arcs [3]. These features allow bounded memory execution and make it is possible to schedule the execution of valid SDF graphs statically i.e. at compile time. Also SDF allows convenient handling of multiple sampling rates in a system. Any node in an SDF graph is enabled for execution when a sufficient number of input tokens is available at the inputs. Thus more than one node may be enabled at any time. This implies that a number of nodes may be fired simultaneously. It is therefore possible to partition an SDF graph into sub-graphs to be scheduled and executed on parallel processors if desired.

Optimal scheduling of SDF graphs can be done in polynomial time for most graphs. Heuristics exist that can achieve or approach the lower bound for program and

data memory requirement. Also the memory and computational resource requirements are known at compile time and are static throughout the execution of the graph [6]

3. TMS320C6x VLIW RISC DSP

Very Long Instruction Word (VLIW) architecture is very popular in the DSP world for several reasons. One important reason is its ability to take advantage of the *Instruction Level Parallelism* (ILP) that is available in typical DSP codes while simplifying the control logic. *Single Instruction Multiple Data* (SIMD) is another way to exploit ILP. However there are advantages of VLIW over SIMD architecture including greater flexibility and reusability of code (a VLIW target does not need major rewriting of the application). Moreover, while compilers perform poorly for traditional DSP's, the compilers for VLIW RISC processors are quite efficient because of Trace Scheduling and Software Pipelining [2]. Compilers do the determination and scheduling of ILP in a high-level language code. Compiler techniques exist for VLIW RISC architectures that allow programmers to write efficient code in a high-level language.

TMS320C6x family is a high-end multimedia VLIW RISC processor family whose architecture and C compiler were designed hand in hand. 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, especially because of its 8 parallel execution units and a very deep pipeline ([7] & [8]). While the compiler achieves high efficiency, for tight loops and other complex algorithms it may be important to harness the full potential of the processor. For this purpose, TI provides optimized assembly code for common DSP structures like FIR, IIR, and FFT etc. It has been reported that these optimizations provide better performance

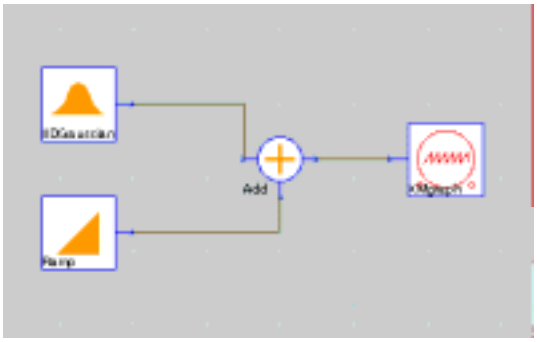
than simple C code. It is our intention to use a library of such optimized code for the *stars* (actors or nodes) in our domain ([5] & [9]).

4. Ptolemy Code Generation Mechanism

Ptolemy's code generation framework uses a methodology wherein code segments pertaining to some functional operations in a particular target language are embedded inside stars as *codeblocks*. Upon execution of the *Universe* (the graph), the code segments from different stars of that universe are stitched together in a sequence (the schedule) generated by the *Scheduler* such that the resulting code can be compiled and run [1].

The initializations (`setup()`), run time (`go()`) and termination (`wrapup()`) methods of Ptolemy's Code Generation (CG) stars differ from simulation stars in that these methods generate code to be compiled and run. CG stars contain two more methods (`initcode()` and `exectime()`). The `setup()` method is called before a schedule is generated and memory is allocated and is used for initializing local variables. The `initcode()` method is called after a schedule is generated is used to generate code before the main schedule loop. The `go()` method generates code for the schedule loop, while the `wrapup()` method generates the code after the schedule loop. More syntactic details are given in [4].

Figure 1 shows the generation of C code from a CGC *Universe* (graph). The universe adds the output of an IID Gaussian source to that of a Ramp source. The schedule and the resulting code are shown in the figure (the initializations and bulk of the IID Gaussian source code are omitted for sake of brevity). It is clear how the *codeblocks* of different stars are stitched together according to the generated schedule.



Schedule for the graph on the right

```
{
  { scheduler "Simple SDF Scheduler" }
  { fire demo.IIDGaussian1 }
  { fire demo.Ramp1 }
  { fire demo.Add.input=21 }
  { fire demo.XMgraph.input=11 }
}
```

… preprocessor directives, like #include's & #define's deleted for brevity …

```
extern main ARGS((int argc, char *argv[]));

/* main function */
int main(int argc, char *argv[]) {
```

… variable initializations deleted for brevity …

… code from initcode() section & default variable initializations deleted for brevity…

```
{ int sdfLoopCounter_8;for (sdfLoopCounter_8 = 0; sdfLoopCounter_8 < 10;
sdfLoopCounter_8++) {
      {  /* star demo.IIDGaussian1 (class CGCIIDGaussian) */
```

… some code for IIDGaussian deleted for brevity …

```
            }
            if (1.0 != 1.0) sum *= sqrt(1.0);
            output_0 = (2.0/3.0) * sum + 0.0;
      }
      {  /* star demo.Ramp1 (class CGCRamp) */
      output_1 = value_5;
      value_5 += 1.0;
      }
      {  /* star demo.Add.input=21 (class CGCAdd) */
      output_2 = output_0 + output_1;
      }
      {  /* star demo.XMgraph.input=11 (class CGCXMgraph) */
    if (++count_6 >= 0) {
        fprintf(fp_3[0],"%g %g\n",
             index_7,output_2);
    }
    index_7 += 1.0;
      }
}} /* end repeat, depth 0*/
```

… code from wrapup() methods deleted for brevity …

```
return 1;
}
```

Figure 1. *Ptolemy CGC Example: a system, its schedule and the generated code*

5. Code Generation for C6x: New CG Domain or a New Target for CGC

The functionality of CGC stars lies in their *codeblocks*. The *codeblocks* contain generic C code that would appear in the final code generated by CGC. For generating optimal code, the *codeblocks* need to be optimized. Writing a new CGC Target for the C6x would use the same generic code and hence the resulting code would not be optimized. This is our motivation for writing an entirely new CG domain for the C6x. Efficient code is essential for the computationally intensive applications for which the C6x is used. Code can be optimized for the C6x either by manipulating the C codes' structure in conformance with the compiler's optimization guidelines or by using C6x Assembly. It has been reported that both these techniques yield some performance improvements over generic C code for most DSP applications.

6. Implementation Strategy

We intend to use C6x's C coding framework instead of assembly, and hence we would derive the CGC6000 from the CGC. The object-oriented design of Ptolemy allows classes for a new domain to be inherited from the already written classes.

We intend to build a C-callable library of functions written in optimized C6x Assembly (or C-code optimized for the C6x). The library would contain functions for common DSP operations. The *codeblocks* in the CGC6000 stars would have calls to these functions. Since the functionality of the stars lies in their *codeblocks*, when linked across this library, optimized executable code would be produced.

The issue of overhead of function calls does arise. For simple stars like add, multiply etc, it is more feasible to inline code instead of making function calls. The

approach of function calls assumes that the functional complexity of the stars is high enough to compensate for the overhead.

For the purpose of this project, we intend to use optimized code provided by TI or other sources. We do not intend to write optimal code ourselves.

We intend to generate code for SDF graphs only. This would allow static scheduling for which we intend to use Ptolemy's schedulers. We are interested in single processor targets only.

7. Results Expected

We intend to test CGC6000 in a number of stages, progressively from simple systems to complex ones. As our final goal, we intend to run the generated code on C6x Evaluation Board, evaluate performance of the generated code for some standard applications and make comparisons with the corresponding code generated by the CGC.

8. References

[1] 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.

[2] 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.

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

[4] Edward A. Lee, et al. University of California at Berkeley, "The Almagest" *Regents of the University of California*, vol. 1-3, 1995.

[5] Paolo Faraboschi, Giuseppe Desoli, Joseph A. Fisher, "The Latest Word in Digital and Media Processing", *IEEE-Signal Processing Magazine*, pp. 59-85, March 1998.

[6] S. Bhattacharya, P. Murthy, E.A. Lee, "*Software Synthesis From Dataflow Graphs*", Kluwer Academic Press, MA, 1996.

[7] S. A. Edwards "*Languages for Digital Embedded Systems*", Kluwer Academic Press.

[8] Prof. Brian L. Evans, "EE382C-9 Embedded Software Systems – Course Notes", Spring 2000.

[9] M. Valliappan et al., "Evaluation of the TMS320C6x C Compiler and Assembly Optimizer; EE382N Superscalar Processor Architecture Project Report", *The University of Texas at Austin*, May 1999.