Presented at the Twenty-Eighth Annual Asilomar Conference on Signals, Systems, and Computers - October 1994

Mapping Multiple Independent Synchronous Dataflow Graphs onto Heterogeneous Multiprocessors

José Luis Pino, Thomas M. Parks and Edward A. Lee

Department of EECS, University of California, Berkeley, CA 94720
{pino,parks,eal}@EECS.Berkeley.EDU

ABSTRACT

In this paper we detail a method to facilitate prototyping of real-time applications on heterogeneous multiprocessors. We introduce a new model of computation that allows for nondeterminate communication between independent dataflow graphs. The graphs may communicate in a manner that does not introduce data dependencies among them. In this paper, we examine the implications of this model, introduce the communication actors required, and discuss scheduling techniques for the multiple independent graphs. We also illustrate this model by detailing some real-time examples that have been constructed in Ptolemy.

1.0 INTRODUCTION

Dataflow is a natural representation for signal processing algorithms. One of its strengths is that it exposes parallelism by only expressing the actual data dependencies that exist in an algorithm. Applications are specified by a dataflow graph in which the nodes represent computational actors, and data tokens flow between them along the arcs of the graph. Ptolemy [1] is a framework that supports dataflow programming as well as other computational models (such as discrete event), in which program graphs have different semantics. Ptolemy allows computation models to be mixed in the specification of complete systems.

Code generation consists of two phases, scheduling and synthesis. In the scheduling phase, the dataflow graph is partitioned for parallel execution. We splice send and receive actors into the graph for interprocessor communication. These actors do the synchronization necessary for a self-timed implementation [2]. For each target processor, an ordering of actor invocations is determined. In the synthesis phase, the code segments associated with each actor are stitched together, following the order specified by the scheduler. Commercial systems which use this "threading" technique include Comdisco's DPC [3] and CADIS's Descartes [4]. The techniques we describe here are complementary to those in DPC and Descartes, and could, in principle, be used in combination with them.

There are several forms of dataflow defined in Ptolemy. In synchronous dataflow (SDF) [5], the number of tokens produced or consumed in one firing of an actor is constant. This property makes it possible to determine execution order and memory requirements at compile time. Thus these systems do not have the overhead of run-time scheduling (in contrast to dynamic dataflow) and have very predictable run-time behavior.

In this paper, we introduce a new model of computation that allows multiple independent dataflow graphs. To allow nondeterminate communication among these graphs, we introduce new communication actors that we call peek and poke. From the perspective of the independent dataflow schedulers, the peek actor is a data source whereas the poke actor is a data sink. In the next section, we will discuss how a heterogeneous target is specified in Ptolemy. In section 3.0, we explain the new model and how it can be used to interconnect independent dataflow graphs. In section 4.0, we study the implications on static and run-time scheduling for the multiple graphs. Finally, we detail two applications that run on a heterogeneous platform consisting of a unix workstation and a DSP card.

2.0 TARGET SPECIFICATION

A key property of Ptolemy that makes specification of heterogeneous targets easier is its use of object-oriented programming techniques. In describing a multiprocessor target, we begin with the specification of each individual processor and build multiprocessor targets hierarchically

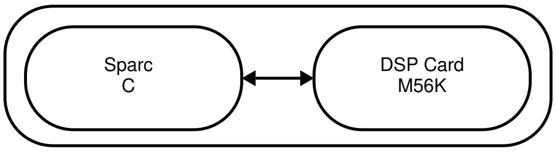

Figure 1. Heterogeneous CGC-S56X target specification.

from these objects. A target specification in Ptolemy manages the flow of the design process; i.e., it defines the methods to schedule the graph, and compile and run the generated code, while taking into account the target resources. A detailed description of the code generation framework in Ptolemy can be found in [6] and with emphasis on single processor targets in [7].

The fundamental building block of a multiprocessor target is a single processor target. In our example, we have an Ariel S-56X card installed in a workstation. An S56X target describes one of the DSP cards. The S56X target specifies the memory resources available on the card and how to download the code into the program memory of the DSP. The S56X target generates assembly code and allocates target-specific resources such as private memory. A CGC target describes the resources of the workstation and generates code in the C programming language. This target is a more general type of target than the S56X target. The code it generates can run on most general-purpose computers.

A multiprocessor target is built from other targets that it contains as children in a hierarchy. These children can be any type of target, from a simple single processor to a complex heterogeneous multiprocessor. The parent multiprocessor target specifies the shared resources and IPC mechanisms of the children (in the form of send/receive and peek/poke actors). A homogeneous multi-DSP target description is detailed in [8].

3.0 MODEL OF COMPUTATION

In this section we define a new model of computation that allows nondeterminate communication among dataflow graphs. We begin by describing synchronous dataflow (SDF) and then define what it means to have multiple independent SDF graphs. We then introduce the peek/poke actors necessary to implement the nondeterminate communication. We have found that this mechanism is very useful for user controls and displays. This model allows a user to specify systems that are largely determinate with manageable nondeterminate

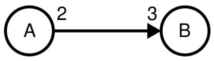

Figure 2. A simple SDF graph.

communication.

3.1 Synchronous Dataflow

Figure 2 shows a simple SDF graph. In this graph, actor *A* produces two tokens and actor *B* consumes three tokens for each firing. In a valid SDF schedule, the first-in/first-out (FIFO) buffers on each arc return to their initial state after one schedule period. Balance equations are written for each arc and an integral repetitions vector is found that solves this system of equations [5]. In this simple example, the balance equation for the arc is: $2 \times R_A = 3 \times R_B$. Thus the repetition vector is: $\begin{bmatrix} R_A & R_B \end{bmatrix} = \begin{bmatrix} 3n & 2n \end{bmatrix}, n \in |Z|$. One of the valid schedules for this graph is *AABAB*. So, given an SDF specification, we can find a schedule at compile-time that is iteratively repeated at run-time.

3.2 Multiple Independent SDF Graphs

In our model, we allow for multiple independent dataflow graphs as shown in figure 3. Here we have two independent graphs communicating nondeterminately over the dotted arcs. These arcs do not add data dependencies. For simplicity, assume that each actor produces/consumes one token on each arc per firing. The schedule for the first graph is *AB*; the schedule for the second graph is *CDE*. One can think of the independent graphs as separate communicating processes.

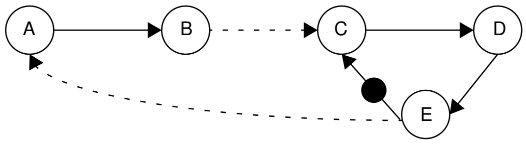

Figure 3. Two independent SDF graphs. The dotted arcs represent communication between the graphs which do not introduce any data dependencies.

3.3 Introducing Nondeterminism: Peek and Poke Actors

For the dotted arcs in figure 3, we splice in *peek* and *poke* actors. This is done in much the same way as send and receive actors are spliced in for self-timed SDF multiprocessor implementations. The peek/poke actors are similar to the send/receive actors except they do not block on the state of their buffer (see figure 4). A send/receive pair has a blocking FIFO buffer between the actors. The send actor will block on a write if the FIFO is full; whereas the receive actor will block on a read if the FIFO is empty. In contrast, the *peek* actor will reuse old data if there is no new data available; whereas the *poke* actor will overwrite old data to make new data available. Thus a *peek* actor appears to be a data source and the *poke* actor appears to be a data sink to the independent SDF schedulers. Furthermore, unlike the SDF use of send/receive as only for interprocessor communication, a peek/poke pair can be used on the same processor.

The various properties of peek/poke actor pairs that we have found useful allow for:

- updating the link at the implicit rate of the graphs
- updating a link at a user-specified rate
- transferring a block of consecutive tokens per update
- optionally aligning the transfers

The first property of the peek/poke actors is how often the link is updated. The simplest configuration for the peek/poke actors is to have the pair updated at the implicit rate of the graphs. Once the peek and poke actors are scheduled, the link will be updated as often as possible given the computed schedules. We have found peek/poke actors useful for separating the user settable run-time parameters from the SDF graph doing the hard real-time computation. Another useful configuration is to have the peek/poke actors updated at a user-specified rate. We have found this useful for updating displays.

The next properties of the peek/poke actors are the number and alignment of data transferred. Transferring a multiple number of tokens per update gives the user a window into a signal. The advantage of controlling the alignment is two fold: it enables the user to download a multiple number of run-time parameters simultaneously; and it enables downloading of a signal (such as a computed FFT) where the alignment is crucial.

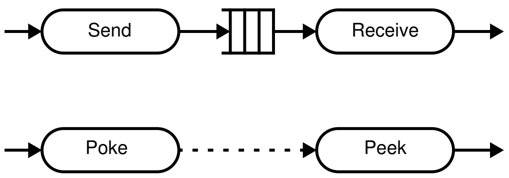

Figure 4. Target spliced communication actors.

4.0 SCHEDULING IMPLICATIONS

In this section, we study the scheduling implications that result from this model. Mapping multiple SDF graphs onto architectures allows for the possibility that two or more independent graphs may be mapped onto a single processor. We assign relative firing rates for each graph to ensure fair scheduling.

Before we discuss methods for scheduling multiple graphs on one processor, we must define what it means to fire a graph. We define a firing of an SDF graph to be one schedule period of an SDF schedule where each of the actors is fired the number of times specified by the minimal repetitions vector. For example, for the graph in figure 2 the schedule given was *AABAB*. One iteration of this schedule would be a firing. However, another valid SDF schedule that does not obey the firing definition is *AABAABAABB*. Here, the schedule is a valid SDF schedule in that it returns the arc to its initial number of tokens. However, the schedule contains more repetitions of each actor than are required.

4.1 Static Scheduling

When we map multiple independent graphs onto a single processor and want to determine the firing order of the graphs at compile-time, we must know the relative rates of a graph. This can either be given explicitly by the user or implicitly one or more of the actors in the independent graphs.

In the explicit case, the user specifies how many times each graph is fired relative to the other. For example, in figure 3, the user could specify that graph *AB* is to fire 4 times for every 3 times graph *CDE* is to fire. So a valid static schedule for the processor is, *4(AB)3(CDE)*.

In the implicit case, the graphs would contain actors that require firing at a certain rate relative to a global clock. For example, if actor *A* reads in samples at 48 kHz and actor *C* reads/writes samples at 44.1 kHz. Here *AB* would have to fire 160 times for 147 times *CDE* is fired. One possible valid static schedule for the processor is, *160(AB)147(CDE)*.

The timing requirements of *AB* may not allow the long delay (147 firings of *CDE*) between the 160th and 161st firings of the graph. The schedule *13(AB)147(ABCDE)* reduces the delay to one firing of *CDE*. The timing requirements may be so strict that not even one complete firing of *CDE* is allowed between firings of *AB*. A more fine-grain interleaving, such as

13(AB)147(ACDBE), may be required in this case.

4.2 Dynamic Scheduling

The relative firing rates of the different graphs may not be known exactly. The timing of graphs *AB* and *CDE* could be controlled by separate hardware clocks which, even though they may have very fine tolerances, will never be exactly synchronized. In this case it is not possible to exactly determine relative firing rates statically, thus requiring dynamic, run-time scheduling.

If the uncertainty in the relative rates is small, for example *AB* fires 160 or 161 times for every 147 firings of *CDE*, then the methods described by Kuroda and Nishitani [10] could be used. A set of 2^{N-1} static schedules is constructed for a N-task system. In this case, the two schedules would be *160(AB)147(CDE)* and *161(AB)147(CDE)*. Measurements from a hardware timer determine which of these schedules is executed next. The major drawback of this approach is that the complexity grows exponentially with the number of tasks.

A more general dynamic scheduling scheme could use a real-time operating system that provides prioritized, preemptive scheduling. By using rate-monotonic priority assignment [9], in which tasks with higher rates are given higher priority, the operating system will provide the necessary run-time interleaving in such a way that all deadlines will be met if possible. If the firing rates and execution times of the tasks are known, then it is possible to guarantee that all deadlines will be met.

Preemptive rate-monotonic scheduling can be useful even with incomplete timing information. Because priorities are fixed, it is possible to predict which tasks will miss their deadlines in an overload situation. Firing rates need not be known exactly as long as there is an ordering of the rates. Execution times, which can be difficult to accurately estimate for programs written in C or other high level languages, are not used in the priority assignments.

Such a solution is attractive for its simplicity, but a large overhead is incurred for the operating system. To avoid this overhead, a simple non-preemptive multitasking kernel can be used with rate-monotonic priorities[11].

5.0 EXAMPLES

In section we detail two simple applications. These examples are running on the heterogeneous platform consisting of a workstation and DSP card described earlier. The applications are synthesized in: C for the workstation; tcl/tk for the workstation graphical user interface; and Motorola 56k assembly code for the DSP card. The first example is an FM synthesis application. This application has five independent SDF graph components. Furthermore, the graphs that are running on the DSP are independent from the graphs running on the workstation. The second example is an acoustic modem. Here there are two independent graphs, where one of the two graphs spans over both the workstation and DSP. Thus this last example also requires the use of send and receive actors.

5.1 FM Synthesis

The FM synthesis graphical specification is shown in figure 5. As was stated above, this application has five independent graphs. The top-level description is shown in the middle of the figure. The subgraph that is enclosed by the dotted line runs of the DSP.

The application specification is hierarchical. Two of the actors have been expanded to expose the hierarchy; all of the actors in the top-level description outside of the

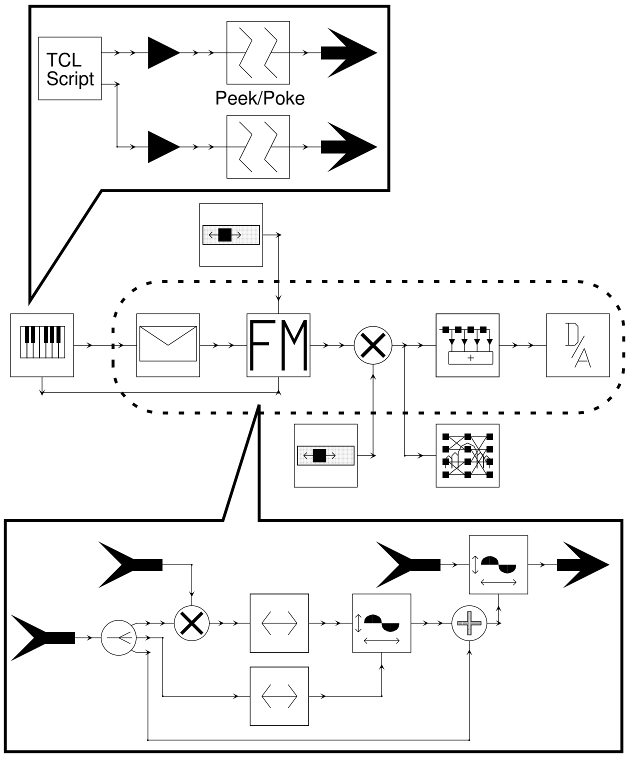

Figure 5. FM sythesis specification. The graph enclosed in the dotted line runs on the DSP card. Peek/Poke actors are spiced on the arcs crossed by the dotted line. The remaining part of the graph runs on the workstation.

dotted arcs expand to graphs similar to the one at the top of the figure. Note the peek/poke actors in this graph. As their icon suggests, they provide the disconnection between the dataflow graphs. These actors are expanded to a peek/poke pair as shown in figure 4. The peek/poke actors disconnect the top subgraph from the subgraph that runs on the DSP. All of the other actors that run on the workstation similarly disconnect themselves from the graph running on the DSP. This allows the DSP subgraph to run as fast as necessary. In the case, the sound is generated at 32 kHz. If we had not disconnected the graph, but rather used send/receive, we would have forced the DSP and the workstation to run in lock step with one another. Thus we would not have been able to run at this high rate.

5.2 Acoustic Modem

The next example which is shown in figure 7 is a simple acoustic modem. A pseudo-random sequence of bits is generated on the workstation. This sequence is sent to DSP which transmits and then receives the bit stream over a speaker/microphone channel. The received bits are then fed back to the workstation, where the errors are displayed to the user (see figure 8).

In this example there are two independent SDF graphs. Note, that one graph here spans over actors that are running on the workstation the DSP. Peek/poke communication schemes could not be used here since we can not afford to loose bits over the channel. The other graph allows the user to adjust the phase of the incoming signal via a slider.

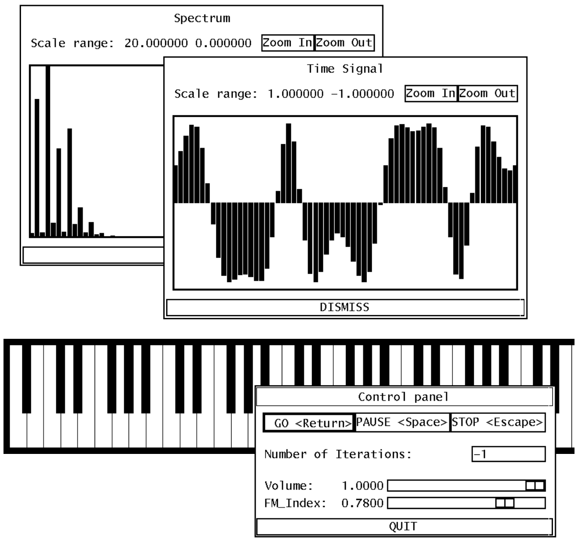

Figure 6. Graphical user interface generated for the FM sound sythesis application.

6.0 CONCLUSIONS

In this paper, we have extended SDF to allow for communication between independent SDF graphs. We have found that the communication mechanism presented is ideal for interfacing user run-time controls and displays. To implement the multiple independent graphs, we introduced new actors called peek/poke which appear to the independent SDF schedulers as source/sink actors. We described some useful properties of the peek/poke actors. We then discussed scheduling implications for mapping independent graphs onto a single processor. In one case we were able to statically schedule the independent graphs; when this was not possible, we showed how rate-monotonic priority assignment could be used in conjunction with a real-time operating system. The model presented is analogous to homogeneous SDF where the repetitions vector is all ones. We are looking at extending this model in the same way that multirate SDF extends homogeneous SDF.

REFERENCES

[1] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation*, special issue on Simulation Software Development, to appear 1994.

[2] E.A. Lee and S. Ha, "Scheduling strategies for multiprocessor real-time DSP.," *GLOBECOM '89*, 1989, p. 1279-83 vol.2.

[3] D.G. Powell, E. A.Lee, and W.C. Newman, "Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams," *ICASSP*, vol. 5, San Francisco, CA, IEEE, 1992, p. 553-556.

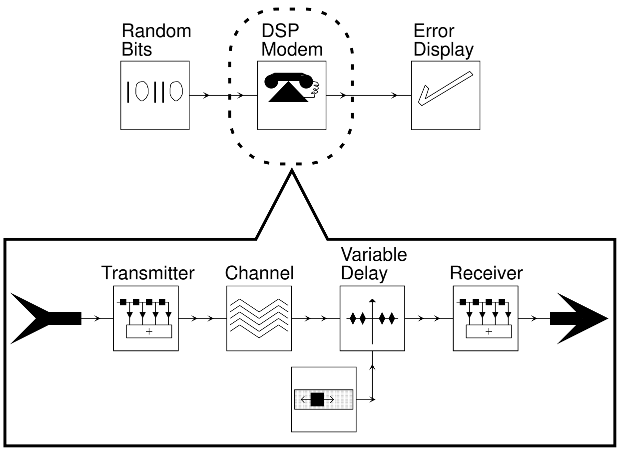

Figure 7. Acoustic modem specification. The dotted line encloses the portion running on the DSP. The modem block expands to the graph shown below.

[4] S. Ritz, M. Pankert, and H. Meyr, "High level software synthesis for signal processing systems," *Proceedings of the International Conference on Application Specific Array Processors*, Berkeley, CA, USA, IEEE Comput. Soc. Press, 1992, p. 679-693.

[5] E.A. Lee and D.G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, 1987, p. 1235-1245.

[6] J.L. Pino, S. Ha, E.A. Lee, and J.T. Buck, "Software Synthesis for DSP Using Ptolemy," *Journal of VLSI Signal Processing*, Synthesis for DSP, 1994, to appear.

[7] J.L. Pino, *Software Synthesis for Single-Processor DSP Systems Using Ptolemy*, Master's Thesis Memorandum UCB/ERL M93/35, University of California at Berkeley, 1993.

[8] S. Sriram and E.A. Lee, "Design and Implementation of an Ordered Memory Access Architecture," *ICASSP*, Minneapolis, MN, IEEE, 1993.

[9] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the Association for Computing Machinery*, vol. 20, no 1., 1973, p. 46-61.

[10] I. Kuroda and T. Nishitani, "Asynchronous Multirate System Design for Programmable DSPs," *ICASSP*, vol 5, San Francisco, CA, IEEE, 1992, p 549-552.

[11] T. M. Parks, E. A. Lee,"Non-Preemptive Real-Time Scheduling of Dataflow Systems," submitted to ICASSP, Detroit, MI, IEEE, 1995.

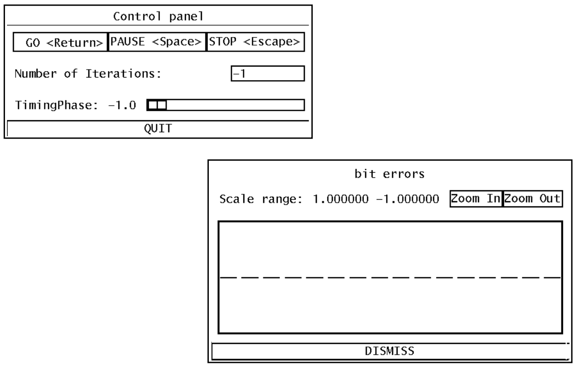

Figure 8. Acoustic modem graphical user interface.