Mapping Multiple Independent Synchronous Dataflow Graphs onto Heterogeneous Multiprocessors

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

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

Abstract

We detail a method to facilitate development 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 between them. We examine the implications of this model, introduce the necessary communication actors, and discuss scheduling techniques for multiple independent graphs. We also illustrate this model with some examples of real-time systems that have been constructed in Ptolemy.

1 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 where the nodes represent computational actors, and data tokens flow between them along the arcs. 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.

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. 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 with 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 and the poke actor is a data sink.

Code generation consists of two phases, scheduling and synthesis. In the scheduling phase, a dataflow graph is partitioned for parallel execution. We splice *send* and *receive* actors into the graph where an arc crosses the boundary between partitions that have been mapped to different processors. These actors do the necessary synchronization to prevent data loss in a self-timed implementation [2]. For each target processor, an ordering of actor invocations is determined. When there are multiple independent dataflow graphs, this scheduling process is repeated for each graph in the system. In the synthesis phase, the code segments associated with each actor are stitched together, following the order specified by the scheduler. Commercial systems that 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.

In the next section, we discuss how a heterogeneous target is specified in Ptolemy. In section 3, we explain the new model of computation and how it can be used to interconnect independent dataflow graphs. In section 4, we study the implications for static and run-time scheduling of multiple graphs. Finally, we present two applications that run on a heterogeneous platform consisting of a unix workstation and a DSP card.

2 Target specification

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

1058-6393/95 $4.00 © 1995 IEEE

Authorized licensed use limited to: University of Pennsylvania. Downloaded on April 17,2024 at 05:56:07 UTC from IEEE Xplore. Restrictions apply.

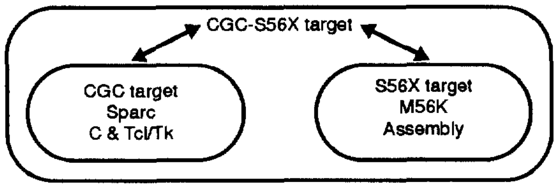

Figure 1. Heterogeneous CGC-S56X target specification.

from these objects. A target specification in Ptolemy manages the flow of the design process: it defines the methods used to schedule the graph, and to 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, shown in figure 1, we have an Ariel S-56X card installed in a workstation. An S56X target describes the DSP card by specifying 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 more general than the S56X target; the code it generates can run on most general-purpose computers.

A multiprocessor target is built by including other targets 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 inter-processor communication mechanisms of the children in the form of send/receive and peek/poke actors. In the example shown in figure 1, the CGC-S56X target is built from single-processor CGC and S56X targets. A homogeneous multi-DSP target is described in [8].

3 Model of computation

In this section we define a new model of computation that allows nondeterminate communication between 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 nondeterminate communication. We have found that this mechanism is very useful for user controls and displays. This model allows a user to specify applications with determinate subsystems that communicate in a nondeterminate fashion.

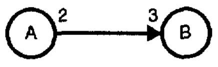

Figure 2. A simple SDF graph.

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 number of tokens in the first-in/first-out (FIFO) buffers of each arc returns to the 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 possible valid schedule for this graph is *AABAB*. So, given an SDF specification, we can find a schedule at compile-time that is iterated at run-time.

3.2 Multiple independent SDF graphs

In our model, we allow 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. The actors A and B are the same as in the previous example, and actors C, D, and E each produce/consume one token on each arc per firing. The schedule for the first graph is *3(A)2(B)*; the schedule for the second graph is *CDE*. One can think of the independent graphs as separate communicating processes.

3.3 Introducing nondeterminism: peek and poke

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

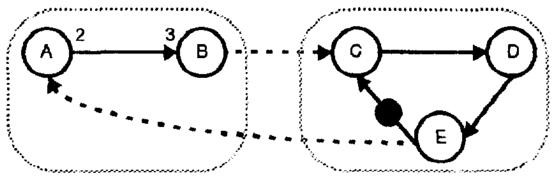

Figure 3. Two independent SDF graphs. The dotted arcs represent communication between the graphs which do not introduce any data dependencies.

Authorized licensed use limited to: University of Pennsylvania. Downloaded on April 17,2024 at 05:56:07 UTC from IEEE Xplore. Restrictions apply.

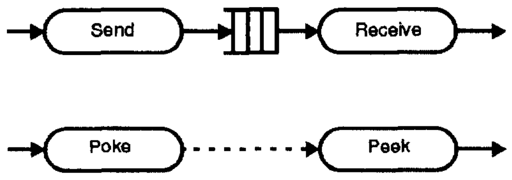

Figure 4. Target spliced communication actors.

pair has a blocking FIFO buffer between the actors. The send actor will block on a write if the FIFO is full, and the receive actor will block on a read if the FIFO is empty. This guarantees that there is no data loss and that old data is not be reused. In contrast, the *peek* actor will reuse old data if there is no new data available, and 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 for interprocessor communication only, a peek/poke pair can be used for either inter- or intra-processor communication.

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

- update the link at the implicit rate of the graphs or at a user-specified rate
- transfer single tokens
- transfer a block of consecutive tokens with a sliding window (see figure 5)
- transfer a block of consecutive tokens aligned to block boundaries (see figure 5)

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 at the rate determined by the execution rates of the independent graphs. 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.

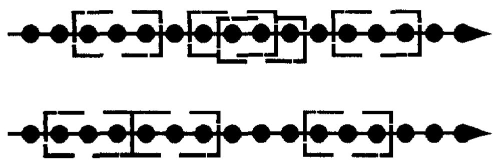

Figure 5. Sliding window transfer (top)
Block aligned transfer (bottom)

The next properties of the peek/poke actors are the number and alignment of data transferred. Transferring multiple tokens per update gives the user a window into a signal. Because there is no synchronization, the window slides by an arbitrary amount. Some tokens could be skipped, or consecutive windows could overlap and tokens would be reused. The advantage of controlling the alignment is two fold: it enables the user to download multiple run-time parameters simultaneously; and it enables downloading of a signal (such as a computed FFT) where the alignment is crucial. Again, because there is no synchronization, some blocks could be skipped and others reused.

4 Scheduling implications

In this section, we study the scheduling implications that result from this model. When multiple SDF graphs are mapped onto a target, it is possible that two or more independent graphs are 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 our definition of a firing 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 graphs. This can either be given explicitly by the user or implicitly by 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 160 times for every 147 times graph *CDE* is to fire. So a valid static schedule for the processor is, *160(3(A)2(B))147(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

Authorized licensed use limited to: University of Pennsylvania. Downloaded on April 17,2024 at 05:56:07 UTC from IEEE Xplore. Restrictions apply.

and actor *C* writes samples at 44.1 kHz. The graph *AB* would have to fire 160 times for 147 firings of *CDE*. One possible valid static schedule for the processor is, *160(3(A)2(B))147(CDE)*.

The timing requirements of graph *AB* may not allow the long delay (147 firings of *CDE*) between the 160th and 161st firings of the graph. The schedule *13(3(A)2(B))147(3(A)2(B)CDE)* 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 graph *AB*. A more fine-grain interleaving, such as *13(3(A)2(B))147(ACADABEB)*, 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(3(A)2(B))147(CDE)* and *161(3(A)2(B))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 bounds on the firing rates and execution times of the tasks are available, 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. Separate stacks must be maintained for the different tasks, and a large amount of context must be saved and restored when switching tasks. To avoid this overhead, a simple non-preemptive multitasking kernel can be used with rate-monotonic priorities[11]. By restricting the points at which a context switch is allowed to occur, a single system-wide stack will suffice and the context that must be saved and restored will be lighter, consisting of just a few machine registers.

5 Examples

In this section we present two simple applications. These examples run on a heterogeneous platform consisting of a workstation and DSP card, as 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 which 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 which has two independent graphs, one of which spans over both the workstation and DSP. Thus this last example requires the use of both peek/poke and send/receive actors.

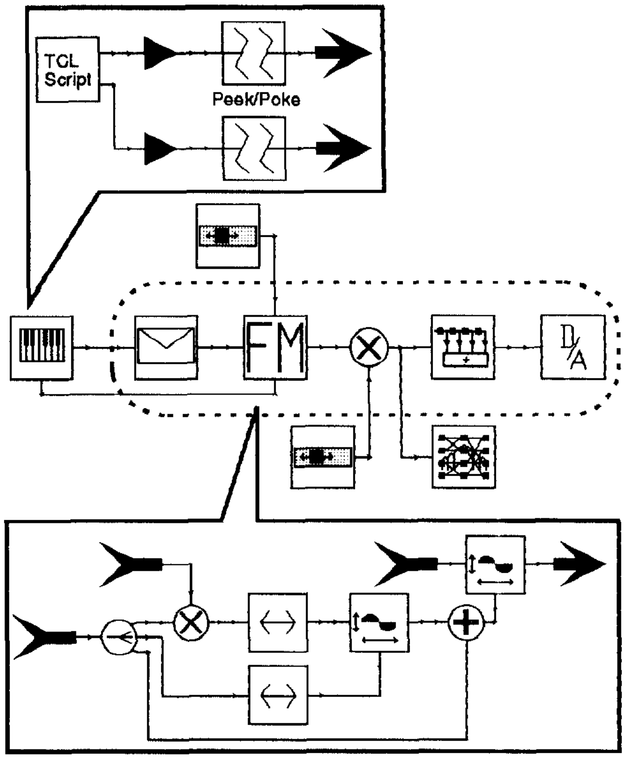

Figure 6. FM sythesis specification. The graph enclosed in the dotted line runs on the DSP card. Peek/Poke actors are spiced on the arcs crossed by the dotted line. The remaining part of the graph runs on the workstation.

Authorized licensed use limited to: University of Pennsylvania. Downloaded on April 17,2024 at 05:56:07 UTC from IEEE Xplore. Restrictions apply.

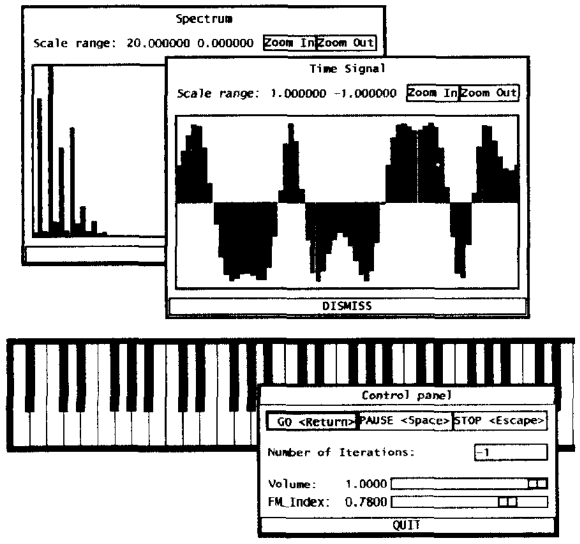

Figure 7. Graphical user interface generated for the FM sound sythesis application.

5.1 FM synthesis

The graphical specification of the FM synthesis example is shown in figure 6. The top-level description is shown in the middle of the figure. This application has five independent graphs, where the one that runs on the DSP is enclosed by a dotted in the figure.

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 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; each is 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 without synchronizing with the workstation. In this case, the sound is generated at a 32 kHz sampling rate. If we had not disconnected the graph with peek/poke actors, but rather used synchronizing send/receive actors, we would have slowed the DSP to run in lock step with the workstation, preventing the DSP from running at the required rate.

5.2 Acoustic modem

The next example, shown in figure 8, is a simple acoustic modem. A pseudo-random sequence of bits is generated on the workstation. This sequence is sent to the DSP, which transmits and then receives the bit stream over

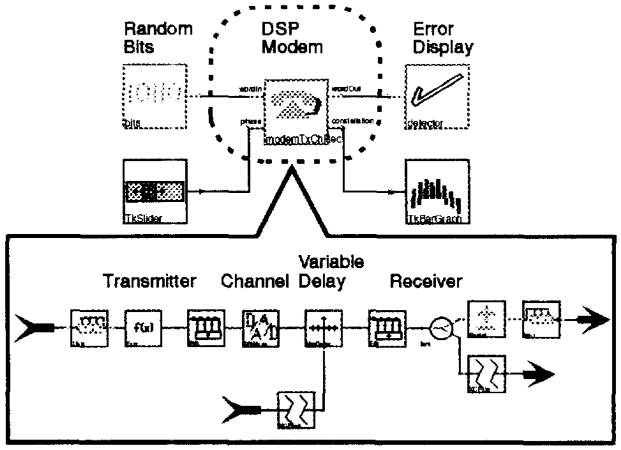

Figure 8. Acoustic modem specification. The dotted line encloses the portion running on the DSP. The modem block expands to the graph shown below.

a speaker/microphone channel. The received bits are then sent back to the workstation, where the errors are displayed to the user (see figure 9).

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

6 Conclusions

In this paper, we have extended SDF to allow for nondeterminate communication between independent SDF graphs. We have found that this communication mechanism is ideal for interfacing user run-time controls and displays to real-time systems. To implement the multiple independent graphs, we introduced new actors called peek and poke which appear to the independent SDF schedulers as data sources or sinks.

The useful properties of the peek/poke actors control the update rate of each link and the size and alignment of

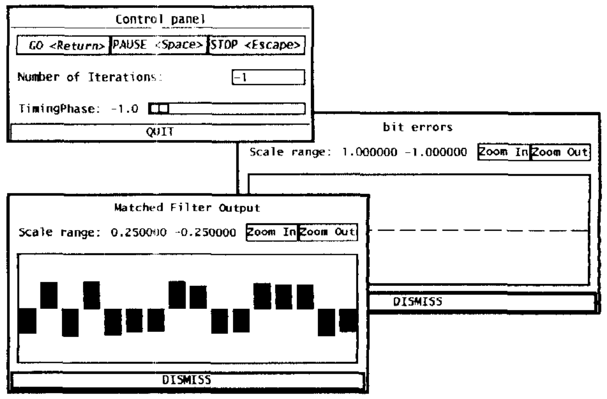

Figure 9. Acoustic modem graphical user interface.

Authorized licensed use limited to: University of Pennsylvania. Downloaded on April 17,2024 at 05:56:07 UTC from IEEE Xplore. Restrictions apply.

each transfer. The rates of the independent graphs determine whether we can use static scheduling or must perform dynamic scheduling. When dynamic scheduling is necessary, rate-monotonic priority assignment can be used in conjunction with a real-time operating system.

We are currently extending this work for both static and dynamic scheduling. For static scheduling, we are exploring the use of hierachical heterogeneous schedulers for distinct subgraphs of the overall specification [12]. To reduce the overhead of dynamic scheduling we are studying prioritized multithreaded execution and non-preemptive rate monotonic scheduling [11].

Acknowledgments

This research is part of the Ptolemy project, which is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), Semiconductor Research Corporation (project 94-DC-008), National Science Foundation (MIP-9201605), Office of Naval Technology (via Naval Research Laboratories), the State of California MICRO program, and the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, Mentor Graphics, Mitsubishi, NEC, Pacific Bell, Philips, Rockwell, Sony, and Synopsys.

José Luis Pino is also supported by AT&T Bell Laboratories as part of the Cooperative Research Fellowship Program.

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.

[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.

[12] J. L. Pino, E. A. Lee, "Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors," submitted to *ICASSP*, Detroit, MI, IEEE, 1995.

Authorized licensed use limited to: University of Pennsylvania. Downloaded on April 17,2024 at 05:56:07 UTC from IEEE Xplore. Restrictions apply.