NON-PREEMPTIVE REAL-TIME SCHEDULING OF DATAFLOW SYSTEMS

Thomas M. Parks and Edward A. Lee

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley CA 94720
{parks,eal}@EECS.Berkeley.EDU

ABSTRACT

Real-time signal processing applications can be described naturally with dataflow graphs. The systems we consider have a mix of real-time and non-real-time processing, where independent dataflow graphs represent tasks and individual dataflow actors are subtasks. Rate-monotonic scheduling is optimal for fixed-priority, preemptive scheduling of periodic tasks. Priority inheritance protocols extend rate-monotonic scheduling theory to include tasks that contend for exclusive access to shared resources. We show that non-preemptive rate-monotonic scheduling can be viewed as preemptive scheduling where the processor is explicitly considered a shared resource. We propose a dynamic, real-time execution model inspired by multithreaded dataflow architectures.

1. INTRODUCTION

Dataflow is a natural model for describing signal processing systems. It is a graphical model of computation where nodes represent computational actors and data tokens flow along the arcs between them. Parallelism is exposed because only a partial order on the actor firings is imposed by the data precedences of the graph topology. Synchronous dataflow (SDF)[1], where each actor consumes and produces a fixed number of tokens on each arc, is especially convenient for describing multirate systems. Because the number of tokens transferred in one firing of an actor is constant, a periodic schedule can be computed statically. For example, the problem of converting from the compact disc (CD) sampling rate of 44.1 kHz to the digital audio tape (DAT) rate of 48 kHz in multiple stages[2] is easy to express with the SDF model.

However, if the CD and DAT have independent clocks then the system must be described as independent tasks (each represented by a separate SDF graph) that communicate through some sort of sample-and-hold mechanism that does not require synchronization, as shown in figure 1. In this example, the first task contains the CD interface and two rate conversion stages. The first stage is implemented with a polyphase FIR filter that consumes 1 token and produces 2, raising the sampling rate from 44.1 kHz to 88.2 kHz. The next stage consumes 3 tokens and produces 4, raising the sampling rate to 117.6 kHz. The sample-and-hold interface to the second task may duplicate or drop samples depending on whether the sampling clock for the DAT is slightly faster or slower than 48 kHz. The discontinuities introduced by this resampling operation are smoothed out by the anti-aliasing filters of the succeeding rate conversion stages.

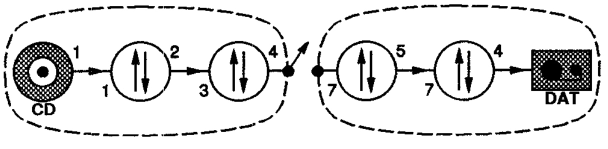

Figure 1: Sample rate conversion from 44.1 kHz to 48 kHz with independent clocks.

Because the relative rates of the clocks in this example are not known exactly, it is impossible to statically determine a valid execution order for the entire system; dynamic, run-time scheduling is required. Signal processing systems that have a mix of periodic, real-time tasks and non-real-time user interface tasks can be described with multiple independent SDF graphs[3], where each dataflow graph is a task and the individual actors in the graph are subtasks that cannot be preempted.

If there is only one task, or if all the tasks are periodic with known relative periods and known execution times, then static scheduling can be used to avoid the expense of dynamic, run-time scheduling. The problem of sequencing a set of tasks with execution times, release times and deadlines on a single processor is NP-compete[4]. There is a large body of literature in the operations research community that deals with this and similar problems. A review of static real-time scheduling techniques is beyond the scope of this paper.

Dataflow schedulers commonly ignore timing constraints. Multi-processor schedulers simply attempt to minimize the duration of one schedule period, which may be adequate for statically scheduled, periodic, real-time systems that do not have strict latency constraints. In a uniprocessor system no such optimization is possible: there is a fixed amount of work to be done and only one processor to do it. But a scheduler could use timing constraints to choose among the execution orders allowed by the data precedence constraints.

0-7803-2431-5/95 $4.00 © 1995 IEEE

Real-time systems commonly use prioritized, preemptive scheduling. Fixed priorities can simplify the implementation of the run-time scheduler and allow predictable, graceful degradation in overload situations: low priority tasks are the first to miss their deadlines. The rate-monotonic priority assignment[5, 6] is an optimal fixed-priority scheme for independent, periodic real-time tasks. Priority inheritance protocols[7] extend rate-monotonic scheduling theory to periodic tasks that are not independent, but must contend for exclusive access to shared resources.

In this paper we apply these results from preemptive scheduling theory to non-preemptive scheduling. The advantages of non-preemptive scheduling are clear. When arbitrary preemption is allowed, large amounts of processor state, possibly including the entire stack, must be saved and restored when one task is suspended and another is resumed. By restricting suspension points to the boundaries between subtasks, the amount of state that must be saved can be reduced significantly.

We propose a run-time scheduler that is derived from a multithreaded dataflow architecture, the threaded abstract machine (TAM)[8]. Early dataflow architectures provided hardware support for fine-grain synchronization where individual machine instructions would be dynamically executed when their operands became available. Because such fine-grain synchronization proved to be a bottleneck, dataflow machines have evolved to support coarse-grain synchronization for threads of sequential instructions. We extend TAM to support static schedules, task suspension, and priorities so that it can be used for real-time systems.

2. RATE-MONOTONIC SCHEDULING

In their landmark paper [5], Liu and Layland address the problem of prioritized, preemptive scheduling for a set of independent, periodic real-time tasks. Each task is characterized by a period, T_i and an execution time C_i. Tasks are executed repeatedly and each invocation of a task must complete before the beginning of the next period. Liu and Layland prove that the rate-monotonic priority assignment, where tasks with a higher rate (shorter period) receive a higher priority, is optimal among all fixed-priority schemes in the following sense:

Theorem 1 *[5] If a feasible priority assignment exists for some task set, the rate-monotonic priority assignment is feasible for that task set.*

A feasible priority assignment guarantees that each invocation of a task completes execution before its deadline at the beginning of the next period. A set of N tasks can be scheduled using rate-monotonic priorities if and only if[6]:

$$\forall i, 1 \leq i \leq N \quad \min_{(k,l)\in R_i} \sum_{j=1}^{i} \frac{C_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil \leq 1$$

where $R_i = \{(k,l) | 1 \leq k \leq i, 1 \leq l \leq \lfloor T_i/T_k \rfloor\}$.

Notice that only the periods T_i, but *not* the execution times C_i, are needed for priority assignment. The execution times C_i are *only* needed to prove feasibility. Because the rate-monotonic priority assignment is optimal, all deadlines will be met *if possible* even if the execution times C_i are not known precisely. Because priorities are fixed, the system exhibits graceful degradation: lower priority tasks are the first to miss their deadlines in an overload situation. Non-real-time tasks can be included in the system by assigning them priorities lower than all the real-time tasks.

3. PRIORITY INHERITANCE PROTOCOLS

In [7] Sha, Rajkumar and Lehoczky address the problem of prioritized, preemptive scheduling for a set of periodic real-time tasks that interact through shared resources. Their motivation is to minimize the effects of priority inversion where a high priority task can be blocked by a lower priority task for an indefinite period of time. This can happen when a low priority task that holds a lock on a shared resource is preempted by one or more medium priority tasks. The high priority task cannot run until the low priority task resumes and releases its lock.

In addition to a period T_i and execution time C_i, each task is characterized by a worst-case blocking time B_i. A set of N tasks is schedulable using priority inheritance if [7]:

$$\forall i, 1 \leq i \leq N \quad \min_{(k,l)\in R_i} \sum_{j=1}^{i-1} \frac{C_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil + \frac{C_i}{lT_k} + \frac{B_i}{lT_k} \leq 1$$

4. NON-PREEMPTIVE RATE-MONOTONIC SCHEDULING

The results from priority inheritance protocols can be used to determine the processor utilization bound for non-preemptive scheduling when the processor is considered as a shared resource. A task acquires and releases the processor for the execution of each subtask. If a higher priority task is ready at the completion of a subtask, the current task will suspend itself. Otherwise it will continue with the next subtask. The worst case blocking time for a task is simply the maximum execution time over all subtasks of lower priority tasks.

$$B_i = \max_{(j,k)\in S_i} C_{j,k}$$

where $C_{j,k}$ is the execution time of the k^{th} subtask of the j^{th} task and $S_i = \{(j,k)|i+1 \leq j \leq N, 1 \leq k \leq N_j\}$ with N_j being the number of subtasks in the j^{th} task.

If there is a non-zero context-switch cost, Δ, then:

$$B_i = 2\Delta + \max_{(j,k)\in S_i} C_{j,k}$$

A higher priority task may become ready just after the decision has been made to activate a different subtask. There is the overhead of switching to that subtask, the execution time of that subtask, and the overhead of switching to the higher priority task before it is allowed to run.

Other work on non-preemptive scheduling[9, 10] lacks the notion of a subtask: once a task begins execution it cannot be suspended and must run to completion. Moitra[11] proposes a scheduling technique *voluntary preemption* that does allow a task to suspend before completion, but his bound

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq \frac{\ln 2}{1+\Delta}$$

is more pessimistic than the one presented here. A different approach to non-preemptive rate-monotonic scheduling was taken in[12]. They addressed the problem of scheduling periodic tasks where the timing is derived from separate hardware clocks, such as the example in figure 1. They use rate-monotonic priorities to construct a set of 2^{N-1} static schedules, where N is the number of tasks, then switch among the schedules at run-time. The complexity of this approach increases exponentially as the number of tasks increases.

5. MULTITHREADED DATAFLOW ARCHITECTURES

Early dataflow architectures executed dataflow graphs directly, where individual machine instructions were dynamically executed when their operands became available. Such machines could not make effective use of pipelines and caches, which have been used very successfully in sequential, von Neumann machines. Thus dataflow machines have evolved to support coarse-grain synchronization for multiple threads of sequential instructions[13].

6. A REAL-TIME THREADED ABSTRACT MACHINE

The threaded abstract machine (TAM)[8] was developed for massively parallel architectures where the goal is to keep processor nodes busy and tolerate long-latency synchronization and communication operations. Enabled tasks are maintained on a ready queue. The currently active task on a processor has a *continuation vector* that holds instruction pointers for enabled subtasks. An activation frame holds local variables that are shared by the subtasks. When a subtask runs, it can enable another subtask by placing its instruction pointer in the continuation vector. In order to maximize locality and take advantage of caches, tasks continue to run until the continuation vector is empty.

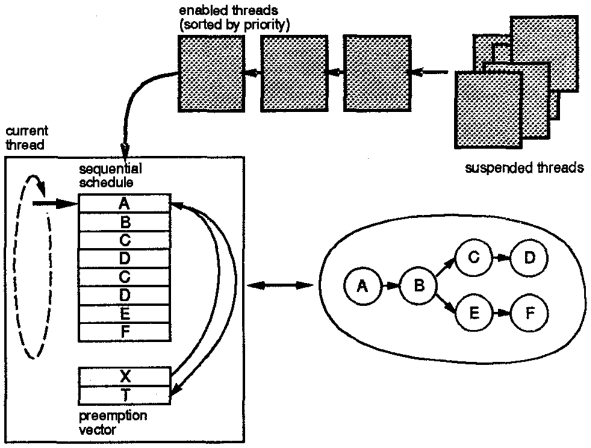

Figure 2: A real-time threaded abstract machine.

We propose several modifications to this execution model to better support real-time scheduling. Figure 2 shows a real-time execution model that is an extension of TAM. Static schedules are allowed in place of the continuation vector. This avoids the overhead of dynamically generating instruction pointers to enable subtasks. The static schedule is cyclicly repeated for periodic tasks. Priorities are assigned to tasks, and the ready queue is sorted by priority. Tasks are allowed to voluntarily suspend and give up control of the processor before completion.

When a high-priority task becomes enabled, a suspension subtask (X) is spliced into the schedule of the currently executing task, and the displaced subtask is stored in a temporary location (T). This is similar to the insertion of a break point. When it executes, the suspension subtask restores the the displaced subtask to its original position in the schedule, saves whatever state is necessary, and transfers control to the highest priority ready task, which is waiting at the head of the ready queue.

Each independent dataflow graph becomes a real-time task. The sequence of actor firings for a task is computed statically. The tasks are scheduled using the non-preemptive rate-monotonic algorithm described in section 4. This execution model could be implemented in software as with TAM, or in hardware as with ⋆T[14] (pronounced *start*) which is essentially a hardware implementation of TAM.

7. CONCLUSION

We have presented sufficient conditions for the existence of a feasible dynamic, non-preemptive rate-monotonic schedule for a set of independent, periodic real-time tasks. We also proposed an efficient dynamic, real-time execution model derived from a multithreaded dataflow architecture.

Preemptive rate-monotonic scheduling is optimal, and in the future we plan to investigate the optimality of non-preemptive rate-monotonic scheduling. We plan to extend our scheduling methods to allow for more general timing constraints and for precedence constraints among tasks.

We will also explore different system representations, such as cyclo-static dataflow[15] and process networks[16]. Cyclo-static dataflow is a generalization of synchronous dataflow that can alleviate the problem where actors with long execution times destroy the feasibility of a non-preemptive, real-time schedule. Having multiple phases of execution, some of which are pure computation with no token consumption or production, reduces blocking times by increasing the number of possible suspension points. Clustering techniques to combine multiple dataflow actors into a single subtask will be explored to solve the opposite problem: increasing the grain size of a fine-grain system to reduce the number of possible suspension points. This can simplify scheduling and reduce overhead. The combination of multi-phase execution and clustering will allow us to adjust the granularity of the graph and trade off between blocking times and scheduling overhead.

8. ACKNOWLEDGEMENTS

This work 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 95-DC-324), National Science Foundation (MIP-9201605), Office of Naval Research (via Naval Research Laboratories), the State of California MICRO program, and the following companies: Bell Northern Research, Dolby, Hitachi, Mentor Graphics, Mitsubishi, NEC, Pacific Bell, Philips, Rockwell, Sony, and Synopsys.

9. REFERENCES

[1] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[2] Ronald E. Crochiere and Lawrence R. Rabiner. *Multirate Digital Signal Processing*. Prentice-Hall Signal Processing Series. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[3] José Luis Pino, Thomas M. Parks, and Edward A. Lee. Mapping multiple independent synchronous dataflow graphs onto heterogeneous multiprocessors. In *Asilomar Conference on Signals, Systems and Computers*, 1994.

[4] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[5] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[6] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.

[7] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[8] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. Tam — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, July 1993.

[9] Xiaoping Yuan and Ashok K. Agrawala. A decomposition approach to non-preemptive scheduling in hard real-time systems. In *IEEE Real-Time Systems Symposium*, pages 240–248, December 1989.

[10] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *IEEE Real-Time Systems Symposium*, pages 129–139, December 1991.

[11] A. Moitra. Voluntary preemption: A tool in the design of hard real-time systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 87–106, January 1992.

[12] Ichiro Kuroda and Takao Nishitani. Asynchronous multirate system design for programmable DSPs. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 549–552, San Francisco, March 1992.

[13] Robert A. Iannucci. *Parallel Machines: Parallel Machine Languages: The Emergence of Hybrid Dataflow Computer Achitectures*. Kluwer Academic Publishers, Boston, 1990.

[14] R. S. Nikhil, G. M. Papadopoulos, and Arvind. $\star$T: A multithreaded massively parallel architecture. In *International Symposium on Computer Architecture*, pages 156–167. The Association for Computer Machinery, May 1992.

[15] Marc Engels, Greet Bilsen, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow: Model and implementation. In *Asilomar Conference on Signals, Systems and Computers*, 1994.

[16] Edward A. Lee. Dataflow process networks. Memorandum UCB/ERL M94/53, Electronics Research Laboratory, University of California, Berkeley, July 1994.