Hierarchical Scheduling and Allocation of Multirate Systems on Heterogeneous Multiprocessors

Yanbing Li
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544, USA

Wayne Wolf
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544, USA

Abstract

This paper describes new algorithms for system-level software synthesis, namely the scheduling and allocation of a set of complex tasks running at multiple rates on a heterogeneous multiprocessor. The tasks may have precedence constraints within them. The multiprocessor may be composed of both programmable and fixed-function processing elements and may have arbitrary interconnect topology. Our hierarchical algorithm takes advantage of the hierarchical structure of the system's task graph to hierarchically allocate and schedule processes on the multiprocessor to meet the hard real-time constraints on the tasks. Multimedia is an important application of our algorithm.

1 Introduction

This paper describes a new, hierarchical algorithm for software synthesis of multi-rate systems on multiprocessors. Starting from a task graph, which describes the process structure of a system which must meet periodic deadlines which occur at several different rates, our algorithm allocates processes to *processing elements (PEs)* and communication to *communication channels* and hierarchically constructs a schedule which guarantees that all the sub-tasks will meet their deadlines. The allocation and scheduling is done interactively. Because the architecture of the multiprocessor is an input to synthesis, our scheduling and allocation algorithm is widely applicable.

Our work was originally motivated by multimedia applications, although the same techniques can be used in the design of systems for other applications. Multimedia is a challenging software synthesis domain because it requires very high performance rates at very low implementation costs. As a result, the available hardware must be utilized very efficiently. Mixed video-audio processing requires multiple, heterogeneous processors to meet the required performance goals. Multimedia systems execute widely varying tasks at widely varying rates.

The design of the system's execution environment is further complicated by the need to *preemptively schedule* some PEs. CPUs in particular are generally allowed to be preempted - a process may be suspended before it is finished so that the PE can execute another process. Preemption increases the efficiency of the multiprocessor but further increases the difficulty of determining whether a proposed schedule is legal. While humans may be able to find good non-preemptive schedules for systems of moderate size, preemption requires the use of CAD tools.

Our algorithm takes advantages of hierarchy in task graphs and successive refinement to heuristically construct a good schedule for the system. The structure of the task graph is used to suggest allocation decisions for processes. We schedule/allocate the processes in a single sub-task and independently schedule the system of sub-tasks. We then combine the fine-grained and coarse-grained schedules to create the system schedule and allocation.

The next section summarizes related work in multiprocessor scheduling. Section 3 describes in detail the problem formulation and describes our algorithm. Section 4 outlines a set of experiments we conducted to evaluate various alternative heuristics for multi-rate system scheduling and allocation on multiprocessors.

2 Previous Work

Several groups have developed scheduling and allocation algorithms for a non-preemptive execution environment. These algorithms usually target a multiprocessor architecture and can handle complex tasks with precedence constraints and data communications. However, they don't support the multi-rate systems. Sih and Lee [1] developed a dynamic level scheduling algorithm which took into account direct communication costs. Their algorithm dynamically recomputes levels for processes during scheduling to take into account how previous decisions have changed the utilization of computation and communication resources. Hoang and Rabaey [3] developed a scheduling/allocation algorithm which takes into account communication costs and can create pipelined implementations. However, their algorithm targets a particular multiprocessor architecture. Pino and Lee [2] developed a hierarchical scheduling algorithm which creates clusters of processes either from the task graph structure or as directed by user input.

In a uniprocessor environment, real-time systems commonly use one of two scheduling policies: *earliest-deadline-first* and *rate-monotonic scheduling* [9]. Both disciplines share common assumptions about execution rates and priorities. They both assume that all tasks are executed periodically. A task is not initi-

ED&TC '97 on CD-ROM

ated until an external signal enables it; once initiated, it must complete by a deadline, which coincides with the end of the period. These scheduling disciplines assign the CPU to a task based on its priority, with the highest-priority active task is allowed to execute on the CPU. Earliest-deadline-first is a dynamic priority scheme: the active tasks are ranked according to the remaining time until their next deadline, with priority inversely proportional to the time to the deadline. Rate-monotonic scheduling is a fixed-priority scheme: the priority of a task depends on its period, with smaller periods rating higher priorities.

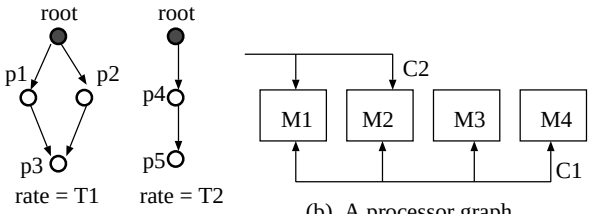
For real-time distributed systems, Peng and Shin [5] developed a branch-and-bound algorithm for allocation of task graphs onto heterogeneous architecture. Because this algorithm is exhaustive in the worst case, it is not practical for large applications. Ramamritham [4] developed a heuristic scheduling and allocation algorithm which took into account data dependencies, communication, and fault-tolerance requirements. However, that algorithm unrolls the tasks to the least-common multiple of the periods of the tasks. Unrolling creates very long schedules; it also eliminates a great deal of information about the structure of the task graph, since multiple executions of a process are treated as separate processes after unrolling. The elimination of structure caused by unrolling a less-than-ideal method for system scheduling; our algorithm performs a partial unrolling which maintains the task structure to guide scheduling and allocation.

Yen and Wolf [10] developed a co-synthesis algorithm which can schedule real-time periodic tasks with data dependencies along with communication cost considered. The algorithm can schedule buses preemptively. Their analysis algorithm uses rate-monotonic scheduling, which we have found to require more preemptions than earliest-deadline-first for the target category of applications. Their synthesis algorithm makes only single-process moves, while our algorithm makes more complex moves to optimize the implementation.

3 Scheduling and Allocation Algorithms

3.1 Problem Specification and Algorithm Overview

The problem specification includes three components: a task graph, a processor graph, and a technology description (Figure 1). As shown in Figure 1-(a), a task graph is a collection subtasks, with each subtask represented by a DAG. Nodes represent processes, which are single execution threads, and directed edges represent data dependencies between processes. Each subtask has a root dummy node, which identifies the initiation point for the subtask. We assume that a process is not enabled to execute until it receives all its data. Each subtask is executed periodically at its specified rate. Many systems exhibit do-all parallelism by executing multiple copies of a single type of subtask-for example, a video decoder may use one task to decode a block in a frame, with a separate subtask for each block in the frame. The task graph may specify that a given subtask is to be replicated n times to cover different instances of a problem; the scheduler can take advantage of this problem structure during scheduling and allocation. The hierarchy between task and subtasks is the basis of the hierarchical algorithm.



(a) A task graph

(b) A processor graph

	M1	M1	M1	M1
p1	10	10	15	20
p2	20	20	30	35
p3	10	10	18	25
p4	50	50	80	80
p5	30	30	40	50

(c) A PE technology description

Figure 1: Problem specification.

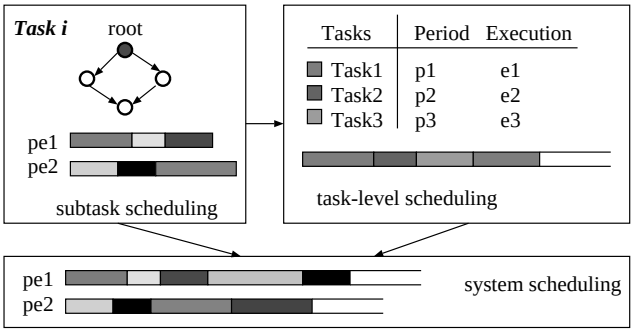


Figure 2: The hierarchical scheduling procedure.

A processor graph, shown in Figure 1-(b), has nodes which represent processing elements and edges which represent communication channels. (Since the channels may connect more than two nodes, this is in fact a hypergraph.) The technology description consists of two maps: the PE technology description, shown in Figure 1-(c), specifies how fast each process runs on each type of PE (not all processes need to be executable on all PE types); the channel technology description specifies the communication time for a unit datum on each type of channel. These times are conflict-free - they do not take into account resource preemption.

As shown in Figure 2, the algorithm hierarchically constructs a schedule for the system by first computing two different schedules. First, it constructs a schedule for the processes in each subtask, assuming that the subtask executes in isolation. Next, it computes a task-level schedule for the subtasks using an earliest-

1. For each process ***node_i*** , calculate **static_urgency (node_i) .**
2. At each **scheduling step**
   for each **ready process** ***node i***
   and each possible choice of PE ***PE_j***
   calculate **dynamic_urgency (node_i, PE_j) .**
   schedule ***node i*** on ***PE_j*** with
   maximum dynamic_urgency value.
   Repeat till all processes are scheduled.

Figure 3: Subtask task scheduling procedure.

```
void subtask_schedule(Subtask S, MultiPE MP, TechInfo T){
 /* calculate initial urgencies of processes */
 foreach node N in S, N.static_urgency = urgency(N);
 /* schedule */
 readynodes = nodes adjacent to root of S;
 while (readynodes != NULL) {
   candidate = NULL; /* the node to schedule next */
   foreach node N in readynodes {
    foreach PE M in MP {
      data_arrival_time = earliest time that all
                          data can arrive at M;
      PE_available_time =earliest time M becomes free;
      delta = median_execution_time(N,T)
              - execution_time(N,M);
      N.dynamic_urgency = N.static_urgency -
        max(data_arrival_time,PE_available_time)+delta;
      if ((candidate == NULL) || (candidate != NULL &&
        candidate.dynamic_urgency <N.dynamic_urgency)){
        candidate = N;
        candidate_PE = M; }}}
   /*schedule candidate and its required communication*/
   schedule_on_channel(candidate,candidate_PE);
   schedule_on_PE(candidate,candidate_PE); }
}
```

Figure 4: The subtask scheduling phase.

deadline-first policy, without looking into the subtask graph, except for using results of step 1 as an estimation for execution time of the subtasks. Finally it creates the final schedule for the system by combining these two types of schedules-the results of the subtask schedules are combined and modified, based on the task-level schedule, to schedule each instance of every subtask on the multiprocessor.

3.2 Subtask Scheduling Phase

The subtask scheduling phase is designed to create an initial allocation and scheduling for the processes in a single subtask for one period. Each subtask is scheduled independently in this step. The cost function of the subtask scheduler is the minimum completion time of a subtask. Our scheduler is similar to that designed by Sih and Lee [1]. Initially, a *static urgency* is calculated for each process based upon the maximum distance of the process to the subtask's end. This is very similar to the priority function that is used in some list schedulers. A subtask is scheduled by selecting a node to be scheduled next and where it will execute on at each *scheduling step*, based upon a *dynamic urgency* (see Figure 3). The *dynamic urgency* is a function

```
RmEdSchedule *task_leve_scheduling(TaskGraph T) {
 /*assign static priority to process based on period*/
 T.sort_periods();   Lcm = least_common_multiple(T);
 task_arrival_time[] = task_arrival_time(T, Lcm);
 /* now adjust initiation times based on EDF */
 t = task_arrival_time[0];   i=0;
 ready_tasks = task_arrivals(t);
 while(ready_tasks!=NULL) {
    if (task T1 running at time t) {
       if ((T1.priority > ready_tasks.head.priority)
         or (T1.deadline < ready_tasks.head.deadline))
          t = task_arrival_time[i++]; /*T1 continues*/
       else {/* T1 is preempted at time t */
         ready_tasks.insert(T1.preempted);
         T2 = ready_tasks.remove_head;
         t =min(task_arrival_time[i++],T2.finish_time);}
     }
    else { /*move a ready process forward*/
       T2 = ready_tasks.remove_head;
       result.schedule(T2, t);
       t=min(task_arrival_time[i++], T2.finish_time);}
    ready_tasks =ready_tasks.append(tasks_arrivals(t));
 }
 return result; }
```

Figure 5: The task-level scheduling phase.

of node and PE, it is calculated for each ready node and each possible choice of PE. The node and PE pair with maximum dynamic urgency value will be scheduled in the scheduling step. *Dynamic urgency* is based upon the static urgency of the node, the communication channel congestion (based upon the data arrival time), the availability of the PE, and the execution speed of the node on the PE. By using *dynamic urgency*, both allocation and scheduling of process node as well as data communication are handled simultaneously and the factor of heterogeneous architecture is taken into consideration.

Unlike some list schedulers, this dynamic urgency is recalculated after every scheduling selection to reflect changes in the system utilization. In order to save computation time, instead of calculating dynamic urgencies for all ready nodes on all possible PEs in each scheduling step, the choice of ready nodes can be limited to only those with high static urgencies and high volume of data (because the combination of these two factors is usually the dominant term in the dynamic urgency), and the choice of PEs can be limited by user specification. Once the most urgent process and its best matched PE is selected, the communication it requires is scheduled onto the channels and the process itself is scheduled onto the selected PE. The subtask scheduling algorithm is shown in Figure 4.

3.3 Task-level Scheduling Phase

Figure 5 outlines the task-level scheduling algorithm. This algorithm separately computes dynamic priorities for the subtasks using an earliest-deadline-first (EDF) policy to preemptively schedule the PEs. A priority is computed for each subtask (all processes in a subtask have the same priority since they run at the same rate). However, the EDF policy is dynamic; the algorithm computes the priority of each subtask

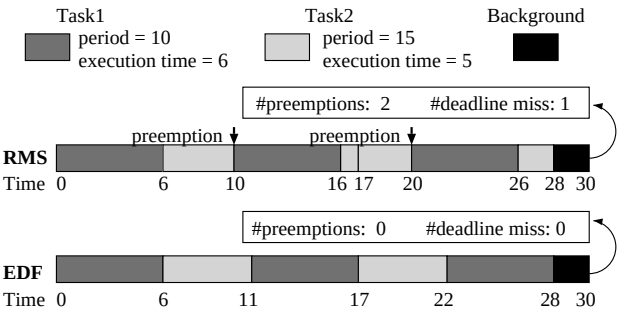


Figure 6: An comparison of rate-monotonic and earliest-deadline-first scheduling.

```
Schedule reschedule_a_subtask(ListSchedule lsch,
                    PE_available_time[] p_at) {
  lsch_copy = lsch.copy;
  while(lsch_copy!=NULL) {
    Process P = lsch_copy.earliest_ready_process;
    (PE,t) = min(PE_available_time[]);
    result.schedule(P, t, PE); }
}
Schedule system_schedule(RmEdSchedule *rms) {
  int PE_available_time[#PE] = {0, 0, ...};
  while (rms!=NULL) {
    RmEdSchedule sch = rms.remove_head;
    Schedule new_sch = schedule_a_subtask(
      sch.task.list_schedule, PE_available_time);
    result.schedule_extend(new_sch); }
}
```

Figure 7: The system scheduling phase.

running on every PE as a function of time. Figure 7 compares earliest-deadline-first scheduling with rate-monotonic scheduling. Figure 6 illustrates the results of these two scheduling policies. With rate-monotonic scheduling, *task 2* is preempted twice by *task 1*. Because of *task 2's* lower priority and the preemption, the first instance of *task 2* misses its deadline. Using EDF, both deadline misses and preemptions are avoided.

We use EDF as the main scheduling policy because it creates a schedule with fewer preemptions than does rate-monotonic scheduling, under the assumption that we have some knowledge about the arrival time of subtasks. If two processes have identical deadlines, then a rate-monotonic policy is used to break the tie; the tie breaker static priority is computed first for each process at the beginning of `task_level_scheduling()`.

3.4 System Scheduling Phase

The complete system schedule is constructed from the results of the subtask and task-level scheduling phases. As shown in Figure 7, this phase moves through the task-level schedule in time order. The system schedule is unrolled to the least-common multiple of the periods. Since each process is only touched once, this scheduling phase is linear in the size of the unrolled system schedule. During unrolling, the task-level schedule is expanded so that each execution of a subtask is represented by the processes and data dependencies of that subtask. As shown in Figure 8, as a subtask instance is expanded, the subtask-level schedule for the subtask is improved based on the current state of the system schedule. A higher-priority process will be moved earlier in time if possible. This may require reallocating that process to a different PE than was originally assigned it during the subtask-level scheduling phase; it may also require moving a lower-priority process backward in time. Due to this repacking of the subtask schedules, different iterations of a subtask may have different schedules and allocations, measured relative to the start of the subtask's period. At the end of this procedure, the complete system schedule has been constructed.

Although the algorithm can construct a preemptive schedule, it tries to minimize the number of preemptions. We use some heuristics to avoid preemption at both task-level and system-level if possible. The task-level scheduling phase used EDF to reduce task preemption. If the arrivals of tasks are deterministic, it also looks ahead to avoid "unnecessary preemptions" (without these preemptions the system can still meet the deadlines). If task-level scheduling result does have preemptions, these preemptions actually are to happen on some process nodes that are running. To reduce the preemptions of individual processes, we use two heuristics. First, a process is not preempted if it is close to finishing and to finish it only uses a small portion of time comparing with the execution time of the preempting process. The thresholds for these decisions is $\delta \times (1 - system\_average\_load)$, in which δ is a parameter set by user. Second, a process is not initiated if it will soon be preempted, as determined by another user-defined parameter. Experiments show that these heuristics usually reduce preemptions while still satisfying the hard deadlines.

The LCM schedule results can be inefficient in case that the task periods are co-prime. However, because the hierarchical scheduling procedure retains some structure in the schedule, instead of enumerating all the instances in LCM, the system schedule can be implemented by constructing tables of priorities as functions of time for the processes in the system. A priority-based real-time scheduler will then select the proper process to run based on this table. This table can be smaller for our algorithm's schedules as compared to schedules constructed by algorithms which fully unroll the task graph.

The computational complexity of this algorithm is smaller than other heuristic scheduling/allocation algorithms because of its hierarchical scheduling heuristics. In the subtask scheduling phase, scheduling of each subtask has complexity of $O(n^2p)$, where n is the number of nodes in the subtask, p is the number of processors. Because the number of nodes in a subtask n is usually small, and sometimes the search for a ready node to schedule a best matched PE is only limited in a subset, the complexity can be reduced further. In the task level scheduling phase, the algorithm treat one subtask as an single node and the worst-case complexity is $O(\#Subtask\_instance \times \#task)$, where the number of subtask instances depends on the num-

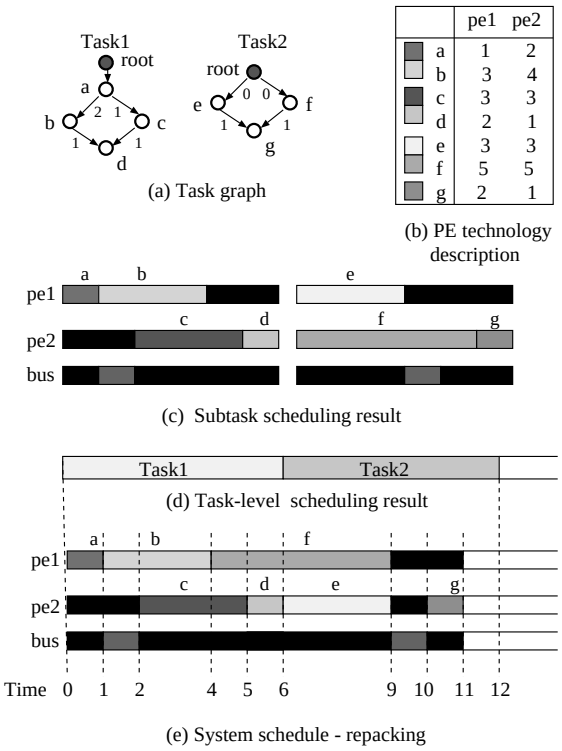


Figure 8: How a subtask's schedule is modified to fit the task-level schedule.

ber of subtasks and the least common multiple of the periods of all subtasks. In the last step, when the algorithm uses the result of phases 1 and 2, the time complexity is linear in the total number of process nodes over all the subtask instances used in task-level scheduling phase, which is $O(\#Subtask\_instance)$.

4 Experimental Results

To test the effectiveness of the algorithm, we have run a number of experiments on some examples, particularly some multimedia applications and obtain promising results. We compare the result quality and the CPU time with some other algorithms.

One of the examples is the MPEG-1 video and audio encoding algorithms. MPEG encoding is very computationally complicated. It involves both intensive computation and large data transfers. Scheduling the encoding algorithms to meet the real-time requirements is challenging. In the video encoder, frames are divided into blocks and the encoding is done for each block. In real time, frames arrive at the rate of 30 frames per second. In our experiment, we assume the frame resolution is 352 240, and the block size is 8×8, which gives 1320 blocks per frame. For the audio encoding, we use the sampling rate of 48 kHz and 16 bits/sample. We assume that the sample data are divided into sections of 1024 samples. Therefore, the rate for audio decoding is $48{,}000/1024 = 46.875$ periods/second. The quantitative characteristics used in our experiment are summarized in Table 1.

Tasks	Audio encoder	Video encoder
Rate	48KH	30 frames/sec
Period(cycles)	3,200,000	5,000,000
# process	5	11/block
# data dependencies	5	15

Table 1: Characteristics of the MPEG-1 video/audio encoder problem.

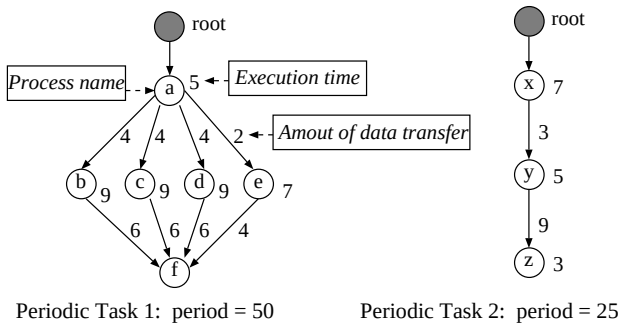


Figure 9: An example used in by Ramamritham in [4]: task graph of two periodic tasks.

Table 2 shows the scheduling result of the MPEG-1 encoding algorithm on 4 identical PEs of 150 MHz, based on the characteristics in Table 1. We connected the four PEs by a common bus. After performing subtask scheduling for both audio and video decoder, we obtained the estimated execution time for each subtask. Task level scheduling is then performed to assign priorities for each subtask. Finally, results from subtask scheduling and task scheduling steps are adjusted to generate the system schedule. The final system schedule successfully schedules both audio and video encoding in real time on 4 PEs.

For this example, we compared our algorithm to traditional fully unrolling approaches by constructing a fully unrolled process graph and applying a list scheduler or our dynamic urgency scheduler to it. The resulting schedule also met the real-time requirements, but the schedule had no obvious structure, which complicates the implementation of the schedule. Scheduling the fully-unrolled process system required about 550 times more CPU time than our algorithm.

We use a small example in [4] as a comparison of the quality of our scheduling results to that of others'. The task graph is shown in Figure 9. As illustrated by Figure 10 and Figure 11, the algorithm in [4] had to use 3 PEs to meet the deadlines while our algorithm only used 2.

To test the scheduler's performance in general, we randomly generated some task sets with random subtask structures and sizes, the we try to schedule them onto 4 or 16 PEs with common buses or arbitrary connections. Table 3 shows the comparison in term of CPU time between our algorithm and the fully-unrolled algorithm. It is clear from the table that our

Subtask	Schedule length / period, 1 PE	Schedule length /period, 4 PEs	Number of periods scheduled
Video encoder	13,400/block, 1320 blocks	4,422,000	25
Audio encoder	21,000	21,000	16

Table 2: Scheduling results of the MPEG-1 encoder problem.

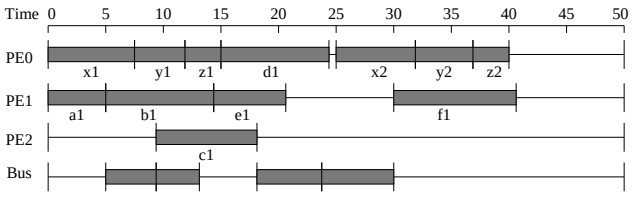


Figure 10: Result given by Ramamritham in [4]: allocation and schedule for the two periodic tasks.

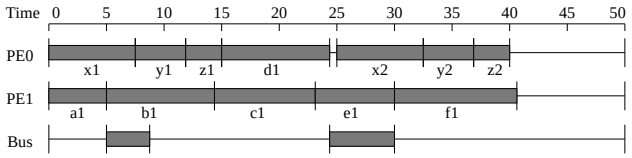


Figure 11: Result given by our algorithm: allocation and schedule for the two periodic tasks.

algorithm runs much faster than the fully unrolling ones. In term of result quality, our algorithm gives feasible schedule to all the 15 random examples that we used.

5 Conclusions

The strength of the algorithm is in the fact that it is much less expensive than existing algorithms and gives effective schedule. This work can be extended in several ways. None of the work of Section 2 considers the effects of communication preemption in shared communication channels such as buses. Those algorithms consider only the transit time of data, not the effects of conflicts with other data transfers on the bus. We plan to incorporate communication preemption analysis into a future version of our algorithm. In addition, we plan to use this algorithm as a component in a co-synthesis methodology for systems implemented on heterogeneous multiprocessors.

As it stands, this algorithm is a very useful tool for the construction of high-performance, multirate systems which execute on multiprocessors. Multimedia is one application domain in which high performance at low cost must be obtained by efficient utilization of the available hardware. Our hierarchical scheme takes advantage of the structure of the specification to efficiently search the solution space. The resulting implementation makes good use of both the processing elements and communication channels in the system. High-quality scheduling/allocation algorithms such as ours allow system designers to concentrate on system-level issues, such as the selection of algorithms.

	Mean	Minimum	Maximum
3 node choice	725.1%	144.9%	1628.0%
all node choice	447.5%	108.9%	1442.3%

Table 3: Speedup of our algorithm in CPU time over fully-unrolled algorithm.

References

[1] G. Sih and E. A. Lee. "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans. Parallel and Distributed Systems,* 4(2), Feb., 1993, pp. 175-187.

[2] J. L. Pino and E. A. Lee. "Hierarchical static scheduling of dataflow graphs onto multiple processors," In Proceedings, *IEEE International Conference on Acoustics, Speech, and Signal Processing,* 1994.

[3] P. D. Hoang and J. M. Rabaey. "Scheduling of DSP programs onto multiprocessors for maximum throughput," *IEEE Trans. on Signal Processing,* 41(4), June, 1993, pp. 2225-2235.

[4] K. Ramamritham. "Allocation and scheduling of complex periodic tasks," In Proceedings, *International Conference on Distributed Computing Systems,* 1990.

[5] D.-T. Peng and K. G. Shin. "Static allocation of periodic tasks with precedence constraints in distributed real-time systems," In Proceedings, *International Conference on Distributed Computing Systems,* 1989.

[6] A. Gerasoulis and T. Yang. "A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors," *J. Parallel and Distributed Computing,* 16, 1992, pp. 276-291.

[7] W. W. Chu and L. M.-T. Lan. "Task allocation and precedence relations for distributed real-time systems," *IEEE Trans. Computers,* C-36(6), June, 1987, pp. 667-679.

[8] L. Sha, R. Rajkumar, S.S. Sathaye. "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems", *Proc. of the IEEE,* Vol. 82, No. 1, pp. 68-82, 1994.

[9] C.L. Liu, J.W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment", *J. ACM,* 20(1), pp. 46-61, 1973.

[10] T.-Y. Yen and W. Wolf. "Communication synthesis for distributed embedded systems," in Proceedings, *ICCAD '95,* IEEE Computer Society Press, 1995.