SOFT REAL-TIME SCHEDULING ON A SIMULTANEOUS MULTITHREADED PROCESSOR

BY

ROHIT JAIN

B.Tech., Indian Institute of Technology, Kharagpur, 2000

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Scienc
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

Abstract

Simultaneous Multithreading (SMT) is a recently proposed technique that seeks to improve processor throughput by allowing the processor to run multiple threads simultaneously. Soft real-time workloads such as multimedia applications are becoming increasingly important and would benefit from the throughput of SMT processors. However, the scheduling of such workloads on SMT processors introduces new challenges that cannot be met by traditional real-time scheduling algorithms. In particular, to run these workloads on SMT processors we must determine 1) which threads to run simultaneously (the co-schedule) and 2) how to share processor resources amongst them, given the real-time constraints of the applications.

We explore several algorithms for co-schedule selection and resource sharing, focusing more on the former. We examine existing multiprocessor scheduling algorithms, including both partitioning algorithms and global scheduling algorithms. We also propose and evaluate some variations on those algorithms that try to utilize the SMT more efficiently.

We find that global scheduling algorithms outperform partitioning algorithms due to their ability to migrate tasks from one context to another, which has no associated cost on an SMT. Variations on the algorithms that try to utilize the SMT more efficiently are effective, helping most for task-sets consisting of tasks with similar utilizations. There is a trade off between algorithmic complexity, profiling overhead, ability to provide admission control and schedulability of the algorithms.

To my parents, bhaiya, bhabhi and Sahil

Acknowledgements

This work has been made possible by the contribution of many people, especially my project partner Christopher J. Hughes and my advisor Sarita V. Adve who guided me through the whole process. Thanks to Jayanth Srinivasan for his valuable ideas and support during our numerous night-outs together. Thanks to Anand Ranganathan and Arun K. Singla for their bright suggestions. Thanks to all the past members of the RSIM group who developed the RSIM simulator and customized the applications which were used in this study.

Table of Contents

1 Introduction 1

2 Resource Sharing Among Co-Scheduled Threads 4

3 Co-Scheduling Algorithms 6
3.1 The Design Space of Co-Scheduling Algorithms 6
3.2 Partitioning Approaches 9
3.2.1 Predicting Execution Time 9
3.2.2 Symbiosis-Oblivious Partitioning 10
3.2.3 Symbiosis-Aware Partitioning 12
3.3 Global Scheduling Approaches 13
3.3.1 Symbiosis-Oblivious Global Scheduling 13
3.3.2 Symbiosis-Aware Global Scheduling 14
3.4 Summary 15

4 Experimental Methodology 16
4.1 Synthetic Workload Evaluation 16
4.1.1 Synthetic Workload Description 16
4.1.2 Metrics and Evaluation 17
4.2 Real Workload Evaluation 18
4.2.1 Real Workload Description 18
4.2.2 Architecture Modeled 18
4.3 Metrics and Evaluation 19

5 Results 21
5.1 Evaluation of Synthetic Workloads 21
5.1.1 Normal Utilization Distribution 21
5.1.2 Bimodal Utilization Distribution 23
5.1.3 Summary 24
5.2 Evaluation of Real Workloads 25

6 Conclusions 30

7 Future Work 31

References 32

1 Introduction

A large part of the recent improvement in processor performance has come from the exploitation of instruction-level parallelism (ILP) in the architecture. High performance general-purpose processors today fetch and decode multiple instructions in parallel, buffer them until their source operands become available, and then issue them to functional units, potentially out of program order. In a modern processor, therefore, a large number of instructions can be in flight at the same time (e.g., 126 instructions in the Intel Pentium 4) – some of these instructions may simply be waiting for their source operands to be generated, others may be in execution, and still others may be awaiting retirement until instructions that are before them in program order are complete. Consequently, processors implement a large number of resources, which are required for peak performance of a single thread, but often have low average utilization over the lifetime of a thread.

Multithreading is a technique proposed to improve resource utilization. Historically, multithreaded processors have been designed to run one thread at any given time – if the current thread experiences a long latency operation, the processor switches to the next thread to hide the latency [1]. In contrast, the recent technique of simultaneous multithreading (SMT) [24] seeks to improve resource utilization even further by allowing the processor to issue (and fetch and decode) instructions from multiple threads *at the same time* (i.e., in the same clock cycle). An SMT processor is very similar to a conventional high-performance processor, but it can additionally dynamically partition its resources among multiple simultaneously active threads on a cycle by cycle basis. This results in high resource utilization and high overall instruction throughputs (albeit possibly with some loss of throughput for an individual thread). These processors, as a result, are already entering a wide range of markets, including network processors [6] and servers [13]; e.g., Intel's Pentium 4 processor (which refers to this technique as hyperthreading).

This paper concerns the use of SMT processors for soft real-time multimedia workloads. Such workloads are becoming increasingly important across a variety of systems, ranging from desktops to handheld devices. In the past, specialized DSP processors and ASICs were often used for these applications. However, with the increasing complexity of these applications and fast changing standards, general-purpose processors are becoming more attractive due to their flexibility, upgradability, compiler and programming support, and time-to-market [3, 7, 9]. These applications often have multiple computationally intensive threads, and are likely to benefit from the increased throughput of SMT processors. For example, a wireless teleconferencing application consists of a video coder, possibly multiple video decoders, speech codecs, and wireless channel codecs. A recent study from industry has explored the use of SMT for wireless phones [14]. SMT processors, however, present new challenges in the scheduling of real-time threads.

In an SMT processor, there are two levels at which scheduling (or resource sharing) decisions need to take place. First, when the number of available tasks[1] is larger than the number of hardware contexts supported by the SMT processor, we need to determine which subset of tasks to co-schedule (i.e., schedule together). Second, we need to determine how to partition processor resources among a set of co-scheduled tasks – in an SMT processor, this partitioning can change each cycle. We refer to the first problem as *co-schedule selection* and to the second problem as *resource sharing*.

The fine-grained resource sharing problem is unique to SMT. Although the general co-scheduling problem must be solved even for conventional multiprocessors, SMT co-scheduling algorithms must consider their interaction with the resource sharing algorithms. In particular, the resource sharing algorithm affects the execution time of each thread by governing the processor resources available to the thread in each cycle. The algorithms with maximum resource utilization would by their nature make decisions on a cycle by cycle basis, involving fine-grained inter-thread interactions that would be hard to predict. Co-scheduling policies that depend on attributes such as task computation times or task utilizations would need to estimate such interactions, and the choice of co-scheduling and resource sharing algorithms may be tightly coupled.

To our knowledge, this is the first study that considers the use of SMT processors for real-time applications. The most widely used SMT resource sharing policy, ICOUNT, seeks to maximize throughput, without consideration of any real-time deadlines [23]. A recent study considered the co-schedule selection problem [21], but was also throughput based for non-real-time tasks and did not consider the resource sharing problem. Raasch and Reinhardt considered SMT resource sharing policies for interactive tasks [20], but not co-scheduling policies. They consider a resource sharing policy that minimizes response time degradation for one interactive task when multiple threads are co-scheduled with it. Their policy gives priority to the interactive task; the others run as "background processes." The response time degradation of the interactive task is minimized, but at the cost of overall system throughput. They conclude that better policies need to be devised. Our work applies to periodic real-time applications, where a task is available each period, and any response time up to the end of the task period (or deadline) is acceptable. Further, we consider soft real-time systems that can afford to miss some deadlines. These differences lead us to develop alternate policies. Finally, there has been some work on scheduling for multithreaded systems where a context switch occurs on a high-latency operation [15]. Since there is no simultaneous multithreading, that work does not have to deal with the resource sharing problem.

This work considers both co-scheduling and resource sharing. We focus on a broader exploration of

[1]We use the terms "task" and "application" interchangeably. We also use the terms "job" and "frame" interchangeably to represent the periodic instances of a task. A "thread" is a job that is currently running on the SMT processor.

co-scheduling, using representative resource sharing policies. We intend to more fully explore the resource sharing design space in the future. We draw on the real-time multiprocessor scheduling literature to consider a variety of co-scheduling algorithms, including both partitioning and global scheduling approaches [16, 8, 2, 18]. We also propose new co-scheduling algorithms that seek to exploit the fine-grained resource allocation ability of SMT processors. We evaluate the algorithms using simulation, with both synthetic and real workloads.

Our results show that in terms of schedulability, the best overall algorithm is a global scheduling algorithm that exploits the SMT processor's fine-grained resource allocation, but also gives priority (while co-scheduling) to tasks with high utilization. This allows the algorithm to take advantage of the SMT without blindly emphasizing throughput over real-time constraints. Unfortunately, the high schedulability comes with two disadvantages: the algorithm does not provide a strict admission control and it requires information on the performance of co-scheduled tasks that may be difficult to acquire. If these disadvantages are serious concerns, then a partitioning policy which avoids use of the fine-grained resource allocation is an alternative. Its performance is generally worse and its algorithmic complexity is higher, but it does provide for admission control and requires performance information only for tasks running in single-threaded mode (as with conventional uniprocessors).

2 Resource Sharing Among Co-Scheduled Threads

An SMT processor is essentially a conventional processor enhanced to be able to process instructions from multiple threads every cycle. The threads share most processor resources, including the instruction fetch mechanism, the instruction window (a buffer for holding in-flight instructions), the execution units (e.g., ALUs), and the caches. While this allows the processor resource utilization to be increased, and thus, overall throughput to be increased, it also means that the performance of each application will most likely be lower when running with other threads than when running alone. Therefore, deciding how simultaneously running threads should share these resources is a critical part of a real-time scheduling algorithm.

Previous work (in non-real-time systems) has focused mostly on resource sharing algorithms to maximize total processor throughput in terms of completed instructions per cycle (IPC) without regard for an individual thread's performance (although some algorithms employ some fairness criteria). In a real-time system, this could clearly have a negative impact on the schedulability of a set of tasks. For example, a high utilization task may be given too few resources to make its deadline regardless of when it begins executing. On the other hand, providing higher total throughput could have a positive impact on schedulability by allowing some tasks to finish executing earlier than they could on a normal uniprocessor. Thus, we consider two classes of resource sharing algorithms – the first class tries to maximize overall throughput while the second tries to guarantee a certain level of performance for one or more threads.

Throughput-driven or dynamic resource sharing. For the throughput driven resource sharing algorithms, we draw on the SMT literature for non-real-time systems. Tullsen et al. considered a number of such resource sharing algorithms [23]. They found the best algorithm was ICOUNT, which gives priority to the thread that has the least instructions in the instruction window (i.e., least number of in-flight instructions) [23]. The intuition is that the thread with the fewest in-flight instructions is making the most progress (since it has fewest instructions stalled in the processor pipeline) and is therefore most likely to effectively utilize system resources. We therefore consider ICOUNT as the representative throughput-driven resource sharing algorithm in this study. In the rest of the paper, we will refer to ICOUNT as the **dynamic** resource sharing algorithm (to contrast with the static algorithm described later).

A drawback of the dynamic algorithm for real-time applications is that performance prediction becomes difficult, because the computation time for a job depends on the behavior of the other co-scheduled jobs. One solution is to profile (measure) performance with each possible set of co-scheduled jobs to obtain performance estimates for future jobs. Such a solution (discussed in more detail in Section 3) introduces additional complexity and overhead for the underlying co-scheduling algorithm.

Resource sharing with performance guarantees. The high throughput of the dynamic resource sharing algorithm comes at the cost of low performance predictability because of its dynamic cycle-by-cycle resource allocation. An alternative is to consider a static resource sharing algorithm where a fixed set of resources is reserved for a given job. This may give sub-optimal resource utilization and throughput; however, the problem of predicting the performance of a job on an SMT is reduced to the problem of predicting its performance on a conventional uniprocessor consisting of only the (reduced) resources reserved for that job. We discuss such a prediction technique in Section 3.

In this work, we consider static resource sharing to represent resource sharing algorithms with performance guarantees. As we will see in Section 3, with static algorithms, the exact basis for allocating a resource reservation to a job is tightly coupled with the co-scheduling algorithm and is therefore discussed along with those algorithms.

Resources controlled by thread-specific resource sharing algorithms. In theory, all processor resources could be controlled by the resource sharing algorithms described above, possibly with different algorithms used for different resources. However, this could result in higher system overhead and complexity. Previous work found that SMT performance was particularly sensitive to the resource control algorithm for the instruction fetch bandwidth [23]. Experiments with our multimedia application suite showed the same effect. Additionally, our experiments also showed senstivity to the algorithm for controlling instruction window allocation (i.e., the maximum number of in-flight instructions allowed for each thread). We found that controlling other processor resources in a thread-blind manner (e.g., priority for execution units is given to the oldest instructions from all threads) did not significantly affect performance of the individual threads. We therefore limit our thread-specific resource sharing algorithms to fetch bandwidth and instruction window allocation. Further, to limit the design space explored, we use the same algorithm for both resources in any one system.

3 Co-Scheduling Algorithms

Section 3.1 describes the attributes we use to characterize and classify SMT co-scheduling algorithms, and summarizes the design space we consider. Sections 3.2 and 3.3 give specific algorithms based on the above attributes. Section 3.4 summarizes the discussion.

3.1 The Design Space of Co-Scheduling Algorithms

Traditional uniprocessor scheduling algorithms assume that only one thread is executing at any given instant. To take advantage of SMT, we must enhance the scheduling algorithm to determine the *co-schedule*, or which threads to run simultaneously. If we consider an SMT processor with N hardware contexts, the co-scheduling algorithm should select up to N tasks out of the set of real-time tasks at any given time to run on the processor simultaneously. We classify the design space of SMT co-scheduling algorithms along two axes described below.

Partitioning vs. Global scheduling algorithms

The SMT co-scheduling problem is similar to the multiprocessor scheduling problem, where at any given time a scheduling algorithm selects up to N tasks to run on N processors. Thus, our first axis for classifying co-scheduling algorithms is analogous to that found in the conventional multiprocessor scheduling literature – *global scheduling* vs. *partitioning*.

In a global scheduling scheme, tasks can be executed on any processor and, if preempted, can be resumed on a different processor. In a partitioning scheme, a partitioning algorithm such as bin-packing is used to bind tasks to processors, and a uniprocessor scheduling algorithm (e.g., Earliest Deadline First (EDF) [17] or Rate Monotonic (RM) [17]) is used on each of the individual processors. Between the two approaches, the partitioning method has received more attention because it provides an admission control and hence, guaranteed run-time performance. The global method, on the other hand, is believed to suffer from scheduling and implementation related shortcomings such as the Dhall effect [8]. However, the scheduling related shortcomings have been alleviated to some extent by algorithms like EDF-US$[\frac{m}{2m-1}]$ [22] , which provide a least upper bound on the system utilization. Also, the implementation related shortcomings such as migration costs are not an issue on an SMT processor (migration on an SMT processor is a degenerate concept since resources can be reallocated across threads with no effort on the part of the hardware).

Andersson et al. [2] present a case for the global scheduling algorithms and Lauzac et al. [16] conclude that global scheduling algorithms outperform partitioning schemes in soft real-time systems, when worst-case computation time of a task is inappropriate. Since we consider soft real-time systems with large

discrepancies between average and worst case times, we consider both partitioning and global scheduling algorithms in this work.

Symbiosis-aware vs. Symbiosis-oblivious algorithms

The second axis of classification does not have an analog in conventional multiprocessor systems. In conventional multiprocessors, the execution time of a job is fixed, regardless of when it is scheduled (ignoring operating system effects). With a dynamic resource sharing algorithm on an SMT processor, the execution time of a job depends on which other jobs are co-scheduled with it. In particular, $Execution\ time = \frac{Number\ of\ instructions \times cycle\ time}{Instructions\ per\ cycle\ or\ IPC}$. With dynamic resource sharing, the IPC (or instruction throughput) of an individual thread (and the overall system) depends on the co-scheduled threads. The best co-scheduling algorithm for an SMT processor may therefore need to consider the interactions of simultaneously running tasks.

Tullsen et al.[23] quantify the notion of "goodness" of a co-schedule consisting of n tasks using a measure called *symbiosis* factor defined as follows:

$symbiosis\ factor = \sum_{i=1}^{n}$(realized IPC of job_i/single-threaded IPC of job_i)

Table 8 gives the symbiosis factors for all 2-task co-schedules for the set of multimedia applications and base system we consider in this study. The higher the symbiosis factor for a set of threads, the more efficiently an SMT processor is being exploited. Ideally, when an SMT processor is simultaneously running N threads, each will have the same performance as when running alone. The symbiosis factor is a measure of how close to this ideal a group of co-scheduled tasks gets. Thus, the second axis of classification for co-scheduling algorithms is symbiosis-aware vs. symbiosis-oblivious algorithms.

To illustrate this with an example, consider 4 tasks τ_1, τ_2, τ_3, and τ_4. Table 1 gives their real-time characteristics and instruction throughputs (IPC) when running simultaneously with one other task. If all these tasks arrive at time T=0, plain EDF scheduling of tasks on a 2-context SMT processor will result in the following sequence of events, where all times are in units of processor cycles:

	τ_1	τ_2	τ_3	τ_4
Period (deadline) in cycles	150	160	150	160
Size (number of instructions)	200	200	200	200
IPC with τ_1		4	2	4
IPC with τ_2	4		2	2
IPC with τ_3	2	4		4
IPC with τ_4	4	2	4	

Table 1 Characteristics of an example real time workload.

Time T=0: Tasks τ_1 and τ_3 are co-scheduled. Both of them run with instruction throughput 2.0. Hence their execution time = $\frac{200}{2.0} = 100$. Both these tasks **meet** their deadlines.

Time T=100: Tasks τ_2 and τ_4 are co-scheduled. Both of them run with instruction throughput 2.0. Hence their execution time = $\frac{200}{2.0} = 100$. Both these tasks **miss** their deadlines.

Now consider a scheduling algorithm which always co-schedules the earliest deadline task with the task which has the most symbiotic relationship with it. If all tasks in our example arrive at time T=0, this scheduling algorithm on a 2-context SMT processor will result in the following sequence of events (again, time is in processor cycles):

Time T=0: Tasks τ_1 and τ_2 are co-scheduled. Both of them run with instruction throughput 4.0. Hence their execution time = $\frac{200}{4.0} = 50$. Both these tasks **meet** their deadlines.

Time T=100: Tasks τ_3 and τ_4 are co-scheduled. Both of them run with instruction throughput 4.0. Hence their execution time = $\frac{200}{4.0} = 50$. Both these tasks **meet** their deadlines.

Hence the algorithm which considers symbiosis successfully schedules a task-set which is unschedulable using EDF. This example illustrates that considering symbiosis in building the schedule can help.

Design space explored.

Figure 1 summarizes the design space we explore for the co-scheduling algorithms. We consider both partitioning and global scheduling approaches. For each approach, we considered EDF or RM as the underlying algorithm (for scheduling within a context in the partitioned approach, and for choosing the next task in the global approach – see Sections 3.2 and 3.3). We only show results with EDF because the results with RM were qualitatively similar. Within each of the partitioning and global scheduling approaches, we consider symbiosis-oblivious and symbiosis-aware algorithms. Within global scheduling, two approaches have been proposed in the past [17, 22] (called PLAIN and US here), as discussed in Section 3.3. We explore the symbiosis-oblivious and symbiosis-aware versions of both approaches. This gives a total of six possibilities so far. Further, for each possibility, there is an option to use a dynamic or static resource sharing algorithm. As discussed in the following sections, the static algorithm seems a promising option for only one of the choices. This gives a total of seven co-scheduling algorithms reported here.

Of the seven algorithms in figure 1, the symbiosis-aware algorithms are particular to SMT processors, and do not have analogs in conventional multiprocessor systems. The partitioning algorithms have analogs in conventional multiprocessor systems, but with SMT processors, there is an additional level of complexity since the resources must also be allocated among the partitions. The following sections discuss these algorithms in more detail.

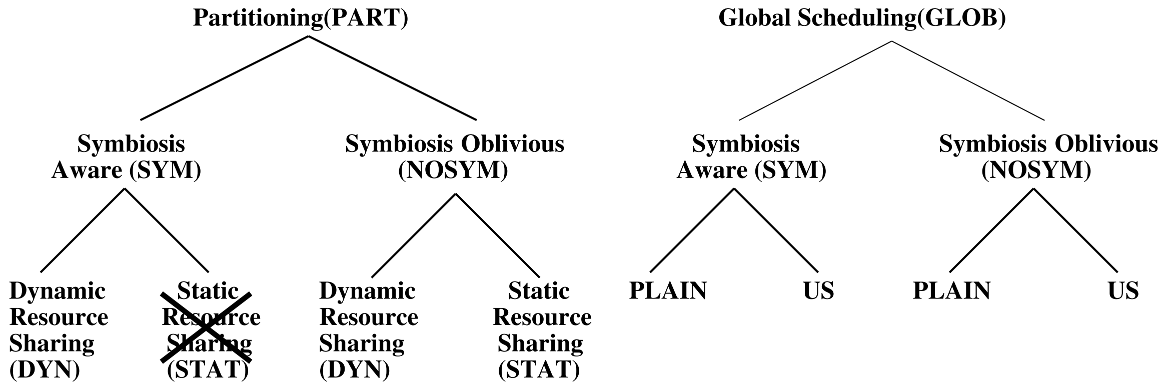

Figure 1 **Design space of co-scheduling policies.**

3.2 Partitioning Approaches

We extend the notion of partitioning in a multiprocessor system to an SMT processor. In a multiprocessor system, a partition is a set of tasks assigned to the same processor. Therefore, no two tasks in the same partition can ever execute simultaneously. Similarly, we consider a partition in an SMT processor to be a set of tasks such that no two will ever execute simultaneously. Unlike in a multiprocessor system, where partitioning avoids overheads such as the cost of physically migrating tasks, partitioning on an SMT processor has the sole benefit that behavior of the tasks in a partitioned system could be more predictable since there are fewer possible interactions.

All of the partitioning algorithms create up to N partitions, where N is the number of contexts supported by the SMT processor. Effective partitioning algorithms need to balance the load on the different partitions, thus requiring the knowledge of execution times of the tasks. On an SMT processor, the partitions need not have equal load since resource allocation need not be the same to all partitions. Regardless, the scheduling algorithm needs to assign an appropriate amount of resources to each partition, and therefore must know the execution times of the tasks. Section 3.2.1 first discusses the issue of predicting execution times and then Sections 3.2.2 and 3.2.3 discuss the specific partitioning algorithms.

3.2.1 Predicting Execution Time

As mentioned earlier, $Execution\ time = \frac{Number\ of\ instructions \times cycle\ time}{IPC}$. To predict the execution time of a job, we need to predict both the number of instructions and IPC for the job. Our previous work has shown that on a conventional uniprocessor, for a given multimedia application, IPC is almost constant across all frames of the same type (e.g., MPEG has I, P, and B frame types) [12]. Therefore, on a conventional

uniprocessor, an IPC estimate for all frames can be obtained by profiling (i.e. measuring) a single frame of each type at the beginning of the application. However, on an SMT processor, the IPC of a task depends on the underlying resource sharing policy and possibly other co-scheduled tasks.

Throughput-oriented resource sharing policies, which dynamically share processor resources among the contexts, make IPC prediction difficult. For such policies, we profile all possible co-schedules in the task-set to obtain the task IPCs. For the multimedia applications we study, we find that the IPC does not change much across different jobs (frames) of the same task for a given co-schedule.[2] The intuition for this is similar to that for our previous observation on IPCs on single threaded processors [10]. Table 8 gives the standard deviation in IPC for each multimedia application we consider in this study across its different frames for all possible 2-task co-schedules, with the dynamic resource sharing policy. Therefore, for each frame type of each application, we profile one frame of that type with all possible co-schedules to obtain IPC estimates for all tasks for all the co-schedules. While this may seem to require a lot of profiling, most multimedia applications run for a large number of frames, making this feasible if not desirable.

With the static resource sharing policy, which allocates a fixed amount of resources to each task, we find that the IPC of a job is dependent only on the resources allocated to it and not on the co-scheduled jobs. For IPC prediction, the scheduler considers the possible (static) resource allocations. For each allocation, the scheduler profiles one job for each task with only the allocated resources in single-threaded mode.

The problem of instruction count prediction is not unique to SMT. Techniques developed to assess execution time in any real-time system would be applicable for instruction count as well and are beyond the scope of this paper. For our experiments, we use profiling to obtain an average instruction count for each task. We profile all tasks for a large number of frames to obtain the average instruction counts. A real system could keep a running average, as done in [5].

3.2.2 Symbiosis-Oblivious Partitioning

For the symbiosis-oblivious partitioning algorithms, we consider bin-packing based algorithms which use the first-fit-decreasing-utilization (FFDU) heuristic [18]. This heuristic sorts the tasks in decreasing order of utilization and allocates a task to the first context which can hold it, as determined using a uniprocessor admission control test. The admission control test used depends on the resource sharing policy, as described

[2] Some applications have different frame types; e.g. MPEG codecs have I,P, and B frames. Strictly speaking, different frame types should be handled separately. Among the real workloads we consider in this study, only MPEG and G728 codecs have multiple frame types. For MPEG codecs, we do not distinguish between different frame types because the dominant frame types (P and B) have IPCs very close to each other. For G728 codecs, the four different frame types appear consecutively in a periodic fashion. For this study, we combine four such frames into one (this is representative of systems that buffer four or more frames).

below. Additionally, for the static resource sharing policy, the partitioning implicitly also does the static resource allocation among the different contexts.

PART-NOSYM-DYN This algorithm employs dynamic resource sharing. Since the measured task utilizations in this case are approximate (Section 3.2.1), we do not have a tight admission control test when trying to assign a task to a partition. We therefore simulate the currently assigned task-set for a hyperperiod. If we find that a partition misses deadlines, we move the smallest utilization task from the violating partition to a partition which meets its tasks' deadlines. A task is never moved into any partition which has ever violated any of its tasks' deadlines. This process is repeated until either all partitions become schedulable, or we can't find a partition to move tasks into, in which case, the algorithm fails. The algorithm converges because tasks can never be moved into the partitions which have ever violated their tasks' deadlines. The complexity of this algorithm is very high, because it involves possibly multiple simulations of a hyperperiod of the task-set. Hence this algorithm may not be practical for real systems. Moreover, this algorithm does not provide a tight admission control because it uses approximate task utilizations. Nevertheless, we study this algorithm to analyze how well the partitioning approaches fare against the global scheduling approaches.

PART-NOSYM-STAT This algorithm employs static resource sharing. Here, we assume knowledge of the task utilizations (as described in Section 3.2.1), and so we have a tight admission test for each partition (EDF in our case). In this case, however, the partitioning algorithm must determine the static resource allocation for each context (the allocation stays fixed for the entire run of the task-set). This is done as follows.

Let $C_1, C_2, \cdots, C_N$ denote the N contexts of the SMT processor. Recall that tasks are assigned to the first available context using the FFDU heuristic and using the EDF admission test as described above. Initially, all resources are allocated to C_1. While assigning a task, let k be such that no tasks are assigned to context $C_i, i > k$. If no context $C_j, j \leq k$ can accomodate the task given the current allocation of resources, resources are re-allocated among contexts $C_j, j \leq k$ such that C_k can accomodate the task. This is done by allocating the minimum amount of resources required by contexts $C_2, C_3, \cdots, C_k$ to remain schedulable and the rest of the resources to C_1. This may cause C_1 to become unschedulable, in which case, we move the smallest utilization tasks from C_1 to the next context which can accomodate the tasks, until C_1 becomes schedulable. If no such context is found, the algorithm fails if $k = N$. Otherwise, the task is assigned to C_{k+1}, and the resource allocation process is repeated. The algorithm converges because tasks are always moved from C_i to C_j such

that $i < j$. The complexity of the algorithm, however, is relatively high, because the EDF admission control has to be performed on a context every time a task is moved from one context to another and whenever the resources are re-allocated among the contexts.

3.2.3 Symbiosis-Aware Partitioning

We also propose and evaluate a symbiosis aware partitioning algorithm that employs dynamic resource sharing. While a symbiosis aware algorithm that employs static resource sharing is possible, we could not clearly identify such an algorithm. Also, using synthetic workload evaluation, we found that with static resource sharing, an algorithm which considers all possible partitions and resource allocations and picks the best does not perform significantly better than the symbiosis oblivious partitioning algorithm. Therefore, we do not consider a symbiosis-aware partitioning algorithm with static resource sharing in this study.

PART-SYM-DYN This algorithm partitions the task-set into at most N partitions, where N is the number of contexts in the SMT processor, such that the average symbiosis among tasks in different partitions is maximized, while the total utilization of tasks in each partition remains reasonably balanced. To achieve this, the algorithm constructs a weighted hypergraph[3] with nodes representing the tasks. The weight on a hyperedge $(u_1, u_2, \cdots, u_N)$ is the inverse of the symbiosis factor of the co-schedule formed by tasks $u_1, u_2, \cdots, u_N$. Each node is weighted with the utilization of the task represented by the node. Since dynamic resource sharing is employed, the task utilizations are approximated as described above. A hypergraph-partitioning algorithm [4] is used to partition the hypergraph into at most N sub-graphs such that the sum of node-weights (i.e., utilizations) in each partition is balanced (we allow up to 20% load imbalance) and the weight of the hyperedges crossing the partitions is minimized (thereby maximizing average symbiosis among the tasks in different partitions). Since the measured task utilizations are approximate, we correct the schedule using simulation in a manner similar to that described for the PART-NOSYM-DYN algorithm. The complexity of the algorithm is very high, for the same reason as for PART-NOSYM-DYN. Hence the algorithm may not be practical for a real system. Moreover, the algorithm does not provide a tight admission control because it uses approximate task utilizations. Nevertheless, we study this algorithm to analyze how well the symbiosis-aware partitioning approaches fare against the global scheduling approaches.

[3] A hypergraph is a graph in which generalized edges (called hyperedges) may connect more than two nodes.

3.3 Global Scheduling Approaches

Global scheduling algorithms do not partition the tasks among the contexts of the SMT processor. These algorithms do not provide a tight admission control. Hence we do not use the static resource sharing policy, which is primarily used to provide an admission control, with these algorithms. We employ only dynamic resource sharing for all the global scheduling algorithms we consider.

3.3.1 Symbiosis-Oblivious Global Scheduling

The symbiosis oblivious algorithms we evaluate are as follows:

GLOB-NOSYM-PLAIN This scheduling algorithm is the earliest deadline first scheduling algorithm (EDF). For an N context SMT processor, the N tasks with the earliest deadlines are chosen as a co-schedule. This algorithm does not require any knowledge of the execution times of the tasks. Hence this algorithm has two benefits – it has a low scheduling complexity and no task information requirement. However, Dhall et al. have shown that task sets with arbitrarily low utilization can be unschedulable with EDF [8]. Specifically, if we have a task-set with a skewed utilization distribution (for example, one task has a much higher utilization than the others), EDF may lead to the high utilization task missing its deadline.

GLOB-NOSYM-US This scheduling algorithm is the EDF-US[$\frac{m}{2m-1}$] algorithm proposed by Srinivasan and Baruah [22]. This algorithm successfully schedules any periodic task system with utilization at most $\frac{m^2}{2m-1}$ on m identical processors. For an N context SMT processor, this algorithm assigns priorities to tasks according to the following rule. If a task T_i has utilization $U_i > \frac{N}{2N-1}$ (where N is the number of contexts), then T_i's jobs are assigned highest priority (ties are broken arbitrarily). All other tasks are assigned priorities according to EDF.

This scheduling approach circumvents the Dhall Effect [8]. For task-sets with a skewed utilization distribution, this algorithm is expected to have higher schedulability than EDF alone because it would assign higher priority to the high utilization tasks and at the same time keep the other tasks from missing deadlines by employing EDF scheduling. For task-sets with a uniform utilization distribution, this algorithm will likely reduce to EDF scheduling alone, because in many cases no task's utilization would cross the set threshold.

In terms of algorithmic complexity, this algorithm is implemented as easily as EDF by simply treating the deadlines of the highest priority tasks as $-\infty$. However, this algorithm assumes the knowledge of

execution times of the tasks, which can be determined as described in Section 3.2.1.

3.3.2 Symbiosis-Aware Global Scheduling

We also propose and evaluate two symbiosis aware global scheduling algorithms, which are extensions of the algorithms described above:

GLOB-SYM-PLAIN This scheduling algorithm extends the EDF scheduling algorithm in the following manner. It first selects the task with the earliest deadline. For the other (N-1) tasks, it chooses the set that maximizes symbiosis when running with the first task. This algorithm seeks to maximize symbiosis while also satisfying real-time constraints by running the EDF-chosen thread. To choose the set of threads to run with the EDF-chosen thread, we use profiling of the possible co-schedules, as explained previously. Therefore, this algorithm statically determines which tasks to run with a given task, independent of the real-time characteristics of the tasks, which can potentially reduce schedulability. We expect this algorithm to perform well when all the tasks in the task-set have similar real-time characteristics (for example, utilization). For such task-sets, there is likely to be some flexibility in when tasks can be scheduled and still make their deadline; thus, ordering the tasks to maximize symbiosis is less likely to trigger missed deadlines. However, for skewed workloads, where giving priority to high utilization tasks is important, this algorithm may not perform as well if the high utilization task is given low priority due to its low symbiosis with other tasks. In terms of algorithmic complexity, this algorithm is only slightly more complex than EDF because it has to identify the tasks with maximum symbiosis with the earliest task. However, it assumes the knowledge of execution times of the tasks (as described in Section 3.2.1) in order to calculate the symbiosis factors.

GLOB-SYM-US This algorithm tries to exploit the benefits of both GLOB-NOSYM-US and GLOB-SYM-PLAIN algorithms. These algorithms are complementary in the sense that GLOB-NOSYM-US yields greater schedulability for task-sets with a skewed utilization distribution by providing high priority to tasks with very high utilization, while GLOB-SYM-PLAIN yields greater schedulability for task-sets with tasks having similar utilizations because it maximizes symbiosis for such task-sets. The GLOB-SYM-US algorithm combines these algorithms as follows.

If a task T_i has utilization $U_i > \frac{N}{2N-1}$ then T_i's jobs are assigned highest priority (ties are broken arbitrarily). All other tasks are assigned priorities according to EDF. If no task has utilization above this threshold, tasks are chosen according to the GLOB-SYM-PLAIN algorithm.

This algorithm is expected to yield high schedulability for task-sets with both uniform and skewed distributions of utilizations on an SMT processor. For task-sets with a skewed utilization distribution, it will give priority to the high utilization tasks, thereby preventing them from missing deadlines. For task-sets consisting of tasks with similar utilizations, it will yield high schedulability by exploiting the fine-grained resource allocation ability of SMT processors in an efficient manner. In terms of algorithmic complexity and knowledge of execution times, this algorithm is similar to GLOB-SYM-PLAIN. Prioritizing high utilization tasks does not add to algorithmic complexity, as discussed above.

3.4 Summary

Table 2 summarizes the co-scheduling algorithms discussed in this section, with their relative algorithmic complexity, admission control ability, and profiling requirements.

	Algorithmic Complexity	**Admission Control Provision**	**Profiling Overhead for Task Execution Times**
PART-NOSYM-DYN	highest	no	high
PART-NOSYM-STAT	high	yes[4]	low
PART-SYM-DYN	highest	no	high
GLOB-NOSYM-PLAIN	lowest	no	none
GLOB-NOSYM-US	lowest	no	high
GLOB-SYM-PLAIN	low	no	high
GLOB-SYM-US	low	no	high

Table 2 Algorithmic complexity, admission control ability, and profiling overhead of the scheduling algorithms.

[4]The admission control provision by the algorithm is similar to that provided by a conventional uniprocessor scheduling algorithm.

4 Experimental Methodology

We evaluate the various scheduling algorithms with simulation of an SMT processor with two hardware contexts (i.e., can execute up to two threads simultaneously). First, we use synthetic workloads to show the algorithms' average performance over a large workload space. Second, we validate the conclusions drawn from the synthetic workload results using a cycle-accurate simulation of some real workloads consisting of video and speech codecs.

4.1 Synthetic Workload Evaluation

4.1.1 Synthetic Workload Description

The synthetic workloads are generated using a methodology similar to that used in [2]. Task-sets are randomly generated and their scheduling is simulated using the various scheduling algorithms on an SMT processor. The total number of tasks, n, follows a uniform distribution with an expected value of $E[n]$ = 8, a minimum of $0.5E[n]$, and a maximum of $1.5E[n]$. The period, T_i, of a task τ_i is chosen from a set 100, 200, 300, 400, 500, $\cdots$, 1600, each number having an equal probability of being selected. The utilization, u_i, of task τ_i, when running in single-threaded mode, follows a normal distribution with an expected value of $E[u_i]$ and a standard deviation of $stddev[u_i] = 0.5E[u_i]$. If $u_i < 0$ or $u_i > 1$, a new u_i is generated. We evaluate workloads with $E[u_i]$ = .15, .20, .25, .30, and .35.

As discussed in Section 3, we expect some algorithms to perform best with workloads consisting of tasks with a skewed utilization distribution. Also, we are concerned with scheduling multimedia applications on an SMT processor. Workloads made up of both video and speech applications tend to have skewed utilization distributions because the video applications are much more computationally intensive. While generating task utilizations in the above manner may allow the representation of some important classes of workloads, it will under-represent workloads with skewed utilization distributions. Hence, we also evaluate synthetic workloads with task utilizations generated using a bimodal distribution. One task, τ_i, in the task set has a utilization which follows a normal distribution with an expected value of $E[u_i] = 0.9$ and standard deviation $stddev[u_i] = 0.1$. If $u_i < 0.8$ or $u_i > 1$, a new u_i is generated. The utilization of any other task, τ_j follows a normal distribution with an expected value of $E[u_j] = 0.03$ and standard deviation $stddev[u_j] = 0.01$. If $u_j < 0$ or $u_j > .1$, a new u_j is generated.

In our simulations, the instruction throughput, or IPC, of task τ_i, when running in single-threaded mode follows a normal distribution with an expected value of $E[IPC_i] = 3.5$ and standard deviation of $stddev[IPC_i] = 1.0$. If $IPC_i < 0$ or $IPC_i > 6$, a new IPC_i is generated. The IPC is constant for

all jobs for that task, as explained in Section 3.2.1. The execution time, C_i, of the task when running in single-threaded mode is computed from the generated utilization of the task and its period. We model tasks with varying job sizes (i.e., varying number of instructions). The size of a job of task τ_i follows a normal distribution with an expected value of $E[I_i] = IPC_i \times C_i$ (where C_i is in cycles) and standard deviation of $stddev[I_i]$. $Stddev[I_i]$ follows a normal distribution with an expected value of $E[stddev[I_i]] = .05$ and standard deviation of $stddev[stddev[I_i]] = .01$. If $stddev[I_i] < .01$ or $stddev[I_i] > .1$, a new $stddev[I_i]$ is generated.

Regarding the interaction of tasks running simultaneously in a co-schedule, we model a dynamic resource sharing policy for synthetic workloads. Each task has a set of instruction throughputs: one for when it is running alone (described above), and one for each of the other tasks in the system, used when it is co-scheduled with one of those tasks. We now describe how we generate the instruction throughtputs for tasks when co-scheduled with another task. For every pair of tasks (τ_i, τ_j), the difference in the instruction throughput of task τ_i when running alone and when co-scheduled with task τ_j (where the latter is $IPC_{i,j}$) follows a normal distribution with an expected value of $E[IPC_i - IPC_{i,j}] = \frac{IPC_i^2}{16}$ and standard deviation of $stddev[IPC_i - IPC_{i,j}] = \frac{IPC_i^2}{32}$. If $IPC_{i,j} < 0$, $IPC_{i,j} > 6$, or $IPC_{i,j} > IPC_i$, a new $IPC_{i,j}$ is generated.

We do not model the static resource sharing policy, and hence do not evaluate PART-NOSYM-STAT with synthetic workloads. Modeling the static resource allocation policy would require a different method for generating the instruction throughputs for co-scheduled tasks. This would make any comparison between PART-NOSYM-STAT and the other algorithms (which use dynamic resource sharing) unfair. We evaluated PART-NOSYM-STAT with synthetic workloads and found the results relative to the other algorithms to be very sensitive to this issue, and therefore do not include these results. However, we do evalute the PART-NOSYM-STAT algorithm for the real workloads.

4.1.2 Metrics and Evaluation

The primary metric used to compare the scheduling algorithms is the *success ratio*, which is the percentage of all generated task-sets successfully scheduled by an algorithm. The success ratio is computed for each type of workload (e.g., expected utilization of 15%, normally distributed) as an average of 2,000,000 task-sets. Since we consider soft real-time systems, a task-set is considered successfully scheduled even if some deadlines are missed. While in a real system each task may have a different tolerance to missed deadlines, in our experiments we assume that all applications are equally tolerant. We evaluate the scheduling algorithms with a tolerance limit of 5% missed deadlines. That is, if any task in a task-set misses more deadlines (during

a hyperperiod simulation) than the limit, the task-set is not considered successfully scheduled. We also ran experiments with a limit of 0% missed deadlines, and the results are qualitatively the same.

4.2 Real Workload Evaluation

4.2.1 Real Workload Description

Our real workloads consist of combinations of eight video and speech codecs. The applications, summarized in Table 3, consist of low-bit rate and high bit-rate speech and video encoders and decoders. We refer to previous work that used these applications for more details on them [10].[5] All applications were compiled with the SPARC SC4.2 compiler with the following options: *–xO4 –xtarget=ultra1/170 –xarch=v8plus*.

Application	IPC	Utilization	Media	Period
GSMenc	5.38	0.17%	Speech	20ms
GSMdec	4.26	0.02%		
G728enc	2.32	3.74%	Speech	2.5ms
G728dec	2.55	2.90%		
H263enc	2.84	32.60%	Video	40ms
H263dec	4.89	1.20%		
MPGenc	2.96	52.40%	Video	66.6ms
MPGdec	4.80	3.50%	Video	33.3ms

Table 3 Application descriptions. The IPC column is the performance in completed instructions per cycle of the application when run alone on the RSIM simulator [19, 11] modeling the processor described in Section 4.2.2 (maximum IPC is 8). The Utilization column is the mean utilization for that application when run alone on the RSIM simulator.

We consider seven different combinations of the applications, given in Table 4. The first workload is a combination of all of the decoders and the second is a combination of all of the encoders. The next four workloads represent configurations of a video teleconferencing system, where a single speech and video encoder are coupled with multiple copies of speech and video decoders. Workload 7 is composed of tasks with similar utilizations, intended to help demonstrate the benefits of the symbiosis-aware algorithms.

4.2.2 Architecture Modeled

The SMT processor modeled is a straightforward extension to a conventional out-of-order superscalar design [23] and is summarized in Table 5. It can simultaneously execute up to two threads. In this study we

[5]Recall from Section 3.2.1 that in this study, one frame of G728enc or G728dec combines four consecutive frames, one of each type.

Workload	Applications
1	1 GSMenc, 1 G728dec, 1 H263dec, 1 MPGdec
2	1 GSMenc, 1 G728enc, 1 H263enc, 1 MPGenc
3	5 G728dec, 1 G728enc, 1 H263enc, 5 H263dec
4	1 GSMenc, 5 GSMdec, 1 MPGenc, 5 MPGdec
5	1 G728enc, 5 G728dec, 1 MPGenc, 5 MPGdec
6	1 GSMenc, 5 GSMdec, 1 H263enc, 5 H263dec
7	6 H263dec, 2 MPGdec

Table 4 Real workload descriptions. Each workload is a combination of applications from Table 3.

assume a perfect instruction cache. The performance of each application running alone on this architecture (using the RSIM simulator [19, 11]) is given in Table 3.

Base Processor Parameters		Base Memory Hierarchy Parameters	
Processor Speed	1GHz	L1 Data Cache	64KB, 2-way associative,
Fetch/Retire Rate	8 per cycle		64B line, 4 ports, 24 MSHRs
Functional Units	6 Int, 3 FP, 4 Add. gen.	L2 Cache	1MB, 4-way associative,
Integer FU Latencies	1/7/12 add/multiply/divide (pipelined)		64B line, 1 port, 24 MSHRs
FP FU Latencies	4 default, 12 div. (all but div. pipelined)	Main Memory	16B/cycle, 4-way interleaved
Instruction window	128 entries	**Base Contentionless Memory Latencies**	
(reorder buffer) size		L1 (Data) hit time (on-chip)	2 cycles
Memory queue size	32 entries	L2 hit time (off-chip)	20 cycles
Branch Prediction	2KB bimodal agree, 32 entry RAS	Main Memory (off-chip)	102 cycles

Table 5 Parameters for the base architecture for real workloads.

4.3 Metrics and Evaluation

To assess the effectiveness of each scheduling algorithm on each task-set, we use a metric called *critical serial utilization*, which is based on the more traditional metric of critical utilization on a conventional (non-SMT) processor. The critical utilization for a non-SMT processor for a task-set with a scheduling algorithm is the total processor utilization obtained by uniformly increasing the utilization of the tasks in the set until a further increase in the utilization of any of the tasks causes the task-set to become unschedulable by the algorithm. We consider a task-set as unschedulable by an algorithm if the fraction of missed deadlines for any task goes above a threshold (5% in our experiments).

For an SMT processor, the notion of critical utilization, or even processor utilization, is not strightforward because the execution time (and hence utilization) of individual tasks depends on the tasks with which they are co-scheduled. We define the notion of *serial utilization* for an SMT processor to be analogous to the utilization of a conventional processor. The serial utilization of a task set $\tau_1, \tau_2, \cdots, \tau_n$ on an SMT

processor is $\sum \frac{C_i}{P_i}$, where C_i is the computation time of task τ_i when run in single-threaded mode on the SMT processor, and P_i is the period of the task. We define *critical serial utilization*, or *CSU*, on an SMT processor for a task-set with a given scheduling algorithm as the total processor serial utilization obtained by uniformly increasing the utilization of the tasks in the set until a further increase in the utilization of any of the tasks causes the task-set to become unschedulable by the algorithm. It is possible for an SMT processor to run a task-set with a CSU greater than 1 since the tasks can overlap with each other.

We evaluate the seven scheduling algorithms from Section 3 with the seven workloads from Table 4 and measure the CSU for each. We do this by reducing the periods of all our applications by the same factor until the task-set becomes unschedulable. The CSU is analogous to the percentage of workloads successfully scheduled in the synthetic workloads evaluation. That is, an algorithm with a higher percentage of successfully scheduled workloads would have a higher CSU for an average workload. This allows us to directly compare results between the synthetic and real workload evaluations.

5 Results

This section presents the results of our experimental evaluation, as discussed in Section 4. Section 5.1 gives the results of the synthetic workload evaluation and Section 5.2 gives the results of the real workload evaluation.

5.1 Evaluation of Synthetic Workloads

5.1.1 Normal Utilization Distribution

We first examine the results with synthetic workloads with utilizations following a normal distribution.

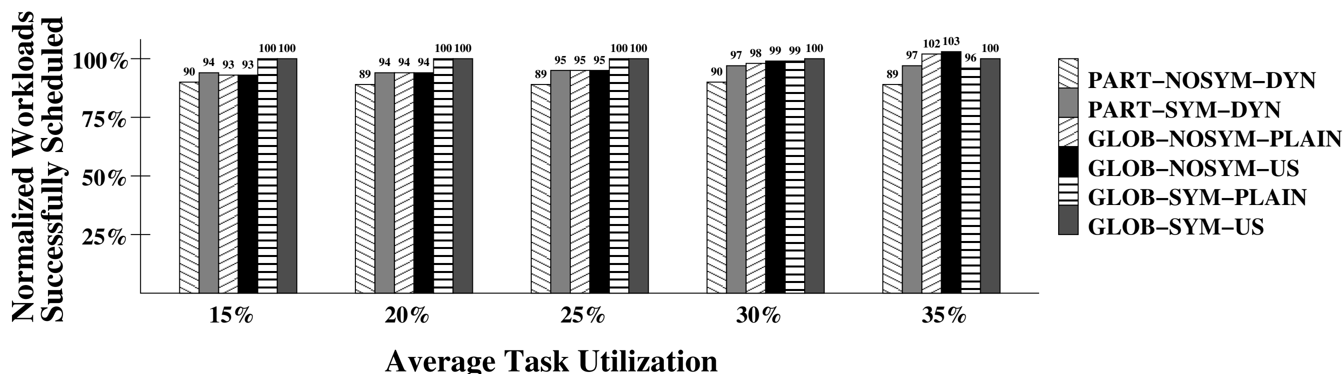

Figure 2 Success-ratios, normalized to the results for GLOB-SYM-US. The task utilizations have a normal distribution.

Average Task Utilization (%)	Success-ratio (%) of GLOB-SYM-US
15	86.4
20	54.0
25	31.4
30	18.8
35	10.1

Table 6 Success-ratio of GLOB-SYM-US with task utilizations normally distributed.

Figure 2 shows the success-ratio of six scheduling algorithms for task-sets with task utilizations normally distributed (expected value ranging from 15% to 35%), as a percentage of the success-ratio of GLOB-SYM-US. Table 6 shows the absolute success ratio of GLOB-SYM-US. Overall, GLOB-SYM-US is the best algorithm in all but one case (where it is only slightly worse than the best). This shows that for the most part, the GLOB-SYM-US algorithm is successful at combining the best features of all choices, as discussed in Section 3.

The global scheduling algorithms are, in general, better than the partitioning algorithms. Since they also have lower algorithmic complexity, this means they have a definite advantage over these partitioning algorithms. For both partitioning and global algorithms, symbiosis-aware algorithms perform better for most cases than symbiosis-oblivious algorithms. Recall however, that this comes at the cost of some increase in algorithmic complexity. We now discuss the effects of each of the design choices described in Section 3 in more detail.

Partitioning vs. Global Algorithms

For each mean task utilization, the best global scheduling algorithms is better than the best partitioning algorithm. This is expected because with partitioning approaches, we lose the ability to migrate tasks among contexts, which can be a hinderance to schedulability in soft real-time systems. This is especially the case on an SMT processor with a dynamic resource sharing policy, because the execution time prediction of tasks can be inaccurate due to the fine-grain interaction of simultaneously running tasks.

For task-sets with mean task utilization below 30%, the best global scheduling algorithms, GLOB-SYM-PLAIN and GLOB-SYM-US, do significantly better than all partitioning algorithms. For task-sets with a mean task utilization of 30%, all global scheduling algorithms do slightly better. For task-sets with a mean task utilization of 35%, all global scheduling algorithms except GLOB-SYM-PLAIN perform better.

Symbiosis Oblivious vs. Symbiosis Aware Algorithms for Partitioning

For the partitioning algorithms, considering symbiosis in constructing the schedule helps significantly in all cases (i.e., PART-SYM-DYN is better than PART-NOSYM-DYN). This, as discussed in Section 3, is expected because maximizing symbiosis between partitions leads to a more effective utilization of the processor resources.

Symbiosis Oblivious vs. Symbiosis Aware Algorithms for Global Scheduling

In general, the symbiosis aware global scheduling algorithms do better than the symbiosis oblivious ones. This is again due to a more effective utilization of the processor by the symbiosis-aware algorithms. As the mean task utilization increases, however, the benefits decrease. At 35% mean utilization, the symbiosis-oblivious algorithms do better than the symbiosis-aware ones. Task-sets with higher total utilization are likely to have less tolerance for algorithms that emphasize throughput over real-time constraints. As discussed in Section 3, trying to exploit symbiosis can hurt schedulability because the real-time constraints of the tasks are ignored for one of the contexts. Maximizing symbiosis for task-sets with smaller total utilization is more beneficial than focusing entirely on real-time constraints. However, as the total task-set utilization becomes large enough, the benefits of higher throughput from a symbiosis-aware policy are outweighed by the penalty of deemphasizing the real-time constraints.

PLAIN vs. US Algorithms for Global Scheduling

Comparing the two symbiosis-oblivious global scheduling algorithms, GLOB-NOSYM-PLAIN and GLOB-NOSYM-US, we see that there is little difference between the two. This is because there is little skew in the task utilizations, making the US algorithm effectively function as the PLAIN algorithm. Task-sets with larger mean task utilization are likely to have more skew, and we do see the US algorithm do very slightly better than the PLAIN algorithm for task-sets with mean task utilization of at least 30%.

Comparing the two symbiosis-aware global scheduling algorithms, GLOB-SYM-PLAIN and GLOB-SYM-US, we observe that there is little difference between them as well, except for the task-sets with 35% mean task utilization. There, GLOB-SYM-US does better because although symbiosis-awareness actually hurts schedulability (as explained above), GLOB-SYM-US also prioritizes high utilization tasks, giving it some of the benefit seen by GLOB-NOSYM-US.

5.1.2 Bimodal Utilization Distribution

We now examine the results with synthetic workloads with utilizations following a bimodal distribution. In these workloads, one task is assigned a very high utilization and the others are assigned relatively low utilizations (as described in Section 4). These workloads are expected to represent real multimedia workloads more closely.

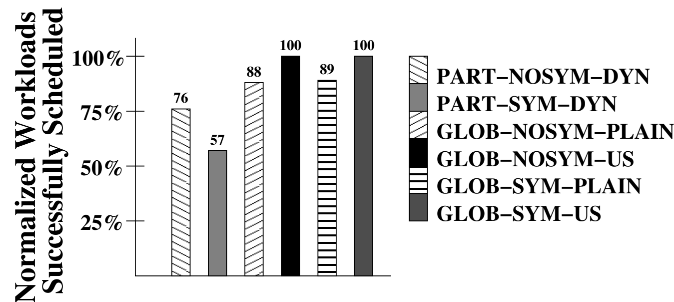

Figure 3 Success-ratios, normalized to the results for GLOB-SYM-US. The task utilizations have a bimodal distribution.

Figure 3 shows the success-ratio of the scheduling algorithms for task-sets with task utilizations bimodally distributed, as a percentage of the success-ratio of GLOB-SYM-US. The absolute success-ratio of GLOB-SYM-US is 32.0%. Again, GLOB-SYM-US performs better than or the same as the other algorithms.

The global scheduling algorithms are all better than all of the partitioning algorithms. Maximizing symbiosis has mixed results for these task-sets; the symbiosis-aware partitioning algorithm does significantly worse than the symbiosis-oblivious one, but symbiosis-awareness has little effect for the global scheduling

algorithms. Among the global scheduling algorithms, prioritizing high utilization tasks(i.e. the US version) provides significant benefits. We now discuss the effects of each of the design choices described in Section 3 in more detail.

Partitioning vs. Global Algorithms

All global scheduling algorithms do significantly better than all partitioning algorithms. All the factors discussed for the task-sets with normally distributed distributions apply here too, but the differences are much more pronounced. This is because for the partitioned approaches, the task-sets with skewed utilization distributions can result in a significant load-imbalance among the contexts. This is not a problem for the global scheduling approaches, which allow task migration among contexts. These results again indicate that global scheduling algorithms are an attractive approach for SMT processors.

Symbiosis Oblivious vs. Symbiosis Aware Algorithms for Partitioning

PART-NOSYM-DYN performs much better than PART-SYM-DYN. For task-sets with a skewed task utilization distribution, load-balancing of partitions becomes the primary concern. Symbiosis-aware partitioning places an emphasis on maximizing the symbiosis among tasks in different partitions, which can exacerbate the load imbalance.

Symbiosis Oblivious vs. Symbiosis Aware Algorithms for Global Scheduling

Symbiosis aware GLOB-SYM-PLAIN and GLOB-SYM-US algorithms are about the same as the corresponding symbiosis-oblivious algorithms, GLOB-NOSYM-PLAIN and GLOB-NOSYM-US. For our bimodal task-sets, this is as expected. As discussed in Section 3, for task-sets such as these, prioritizing the high utilization tasks is more important than scheduling based on symbiosis, which can even hurt schedulability.

PLAIN vs. US Algorithms for Global Scheduling

The global scheduling algorithms that give priority to high utilization tasks, GLOB-NOSYM-US and GLOB-SYM-US, do significantly better than those that do not. For our bimodal task-sets, this is expected when using the US algorithm, as explained above. Also note that GLOB-SYM-US performs as well as GLOB-NOSYM-US for the bimodal task-sets. Hence, we observe that GLOB-SYM-US succeeds in exploiting the benefits of prioritizing high utilization tasks.

5.1.3 Summary

Table 7 summarizes the effects of the different SMT scheduling algorithm design decisions for both classes of task-sets studied. Overall, GLOB-SYM-US is the best choice as it performs better than all other algorithms in all but one case. In this case, where the total task-set utilization is high, GLOB-SYM-US does only

slightly worse than the best algorithm. The second most attractive algorithm is GLOB-NOSYM-PLAIN, due to its lower complexity. However, this lower complexity comes at the cost of lower performance than GLOB-SYM-US in almost all cases.

Decision	**Impact on schedulability**
Partitioning vs. Global	Global scheduling is better
Symbiosis Oblivious vs. Symbiosis Aware	Symbiosis-aware algorithms are better for task-sets with similar task utilizations, except when the total task-set utilization is very large, in which case they are worse. For task-sets with skewed task utilizations, symbiosis-awareness hurts performance for partitioning algorithms, but has little effect on global algorithms.
PLAIN vs. US for global scheduling	Prioritizing high utilization tasks has little effect for task-sets with similar task utilizations, but is significantly better for task-sets with skewed task utilizations.
Best combination	The best combination is GLOB-SYM-US – Global scheduling with symbiosis awareness and prioritization of high utilization tasks.

Table 7 Summary of impact of SMT scheduling algorithm design decisions on schedulability.

5.2 Evaluation of Real Workloads

We now examine the results with the real workloads described in Section 4.2, and compare these results to those from the evaluations of synthetic workloads.

Table 8 gives the average IPC and standard deviation in IPC across frames for each multimedia application we consider in this study, for all possible 2-task co-schedules, with the dynamic resource sharing policy. It also gives the average total IPC and the symbiosis factor for each co-schedule. Some entries in the table are omitted because for the corresponding pairs of applications, at least one of the applications couldn't complete even one frame before the other finished.

Figure 4 gives the IPC of each multimedia application used in this study with different resource allocations. Resource allocation n corresponds to n fetch slots and $16n$ instruction window entries. This profile information is used to predict the IPC of an application for a given resource allocation when using the static resource sharing policy.

Figure 5 shows the CSU (as defined in Section 4.2) of each scheduling algorithm for each workload discussed in Section 5.2. We find that the high-level conclusion from the synthetic workload evaluation still holds – GLOB-SYM-US always performs better than or the same as all the algorithms that use dynamic resource sharing. Compared to GLOB-NOSYM-PLAIN, which has the least algorithmic complexity of the algorithms employing dynamic resource sharing, GLOB-SYM-US is better in almost all cases, but the difference is less than for the synthetic workloads. These results also include an algorithm which employs static resource sharing (PART-NOSYM-STAT). PART-NOSYM-STAT does slightly better than GLOB-SYM-US

App 1	App 2	Avg. IPC of App 1	Avg. IPC of App 2	Avg. IPC of the coschedule	Symbiosis factor of the coschedule	Std.Dev. in IPC of App 1	Std.Dev. in IPC of App2
H263enc	MPGenc	2.2	2.7	4.9	1.70	3.7	4.9
H263enc	GSMenc	2.6	3.3	5.9	1.54	0.3	0.4
H263enc	G728enc	2.5	2.1	4.6	1.79	2.1	0.8
G728enc	MPGenc	1.9	3.1	5.0	1.86	2.2	9.5
G728enc	GSMenc	2.0	3.9	5.9	1.60	1.2	1.9
GSMenc	MPGenc	2.9	2.9	5.9	1.54	–	–
H263dec	MPGdec	3.4	3.2	6.6	1.38	2.3	3.6
H263dec	GSMdec	3.9	2.5	6.3	1.39	2.6	0.8
H263dec	G728dec	4.0	2.1	6.1	1.65	2.4	0.8
G728dec	MPGdec	2.1	3.8	6.0	1.65	0.8	4.3
G728dec	GSMdec	2.2	2.8	4.9	1.50	1.0	1.3
GSMdec	MPGdec	2.5	3.8	6.3	1.39	1.6	3.6
H263enc	H263dec	2.3	3.5	5.9	1.56	2.8	1.4
H263enc	MPGdec	2.4	3.4	5.8	1.56	2.1	0.5
H263enc	G728dec	2.4	2.2	4.7	1.73	2.2	0.3
H263enc	GSMdec	2.6	2.9	5.5	1.59	–	–
MPGenc	H263dec	2.6	3.3	5.9	1.57	6.1	4.1
MPGenc	MPGdec	2.7	3.2	5.8	1.57	5.1	4.7
MPGenc	G728dec	2.9	2.0	5.0	1.79	7.5	2.3
MPGenc	GSMdec	2.5	2.7	5.2	1.47	–	–
G728enc	H263dec	2.0	3.9	5.9	1.67	1.1	1.8
G728enc	MPGdec	2.0	3.9	5.9	1.68	1.0	4.0
G728enc	G728dec	2.0	2.1	4.1	1.69	1.2	1.1
G728enc	GSMdec	2.0	2.8	4.8	1.52	1.6	1.0
GSMenc	H263dec	3.1	3.5	6.7	1.32	1.5	1.7
GSMenc	MPGdec	3.1	3.5	6.6	1.33	2.1	3.7
GSMenc	G728dec	3.9	2.1	6.1	1.57	1.8	1.0
GSMenc	GSMdec	3.8	2.6	6.4	1.32	1.6	1.2
H263enc	H263enc	2.4	2.4	4.9	1.72	1.4	1.4
MPGenc	MPGenc	2.3	2.3	4.6	1.57	4.1	4.0
G728enc	G728enc	2.0	2.0	3.9	1.68	1.1	1.1
GSMenc	GSMenc	3.3	3.3	6.6	1.23	1.5	1.5
H263dec	H263dec	3.3	3.3	6.6	1.38	1.5	1.3
MPGdec	MPGdec	3.1	3.2	6.3	1.34	4.0	4.0
G728dec	G728dec	2.1	2.1	4.3	1.67	1.1	1.1
GSMdec	GSMdec	2.7	2.7	5.5	1.28	0.3	0.3

Table 8 Average IPC, Symbiosis factor, and Standard Deviation in IPC across frames for all pairs of multimedia applications with the dynamic resource sharing policy.

in two cases, but worse in all others. While PART-NOSYM-STAT provides admission control and requires less profiling than GLOB-SYM-US, its performance is generally worse due to its lower throughput from static resource sharing and from its inability to migrate tasks across contexts. Therefore, if profiling or admission control is a key concern, PART-NOSYM-STAT is the best algorithm; otherwise, GLOB-SYM-US is best. Next we discuss each workload in more detail.

Workload 1

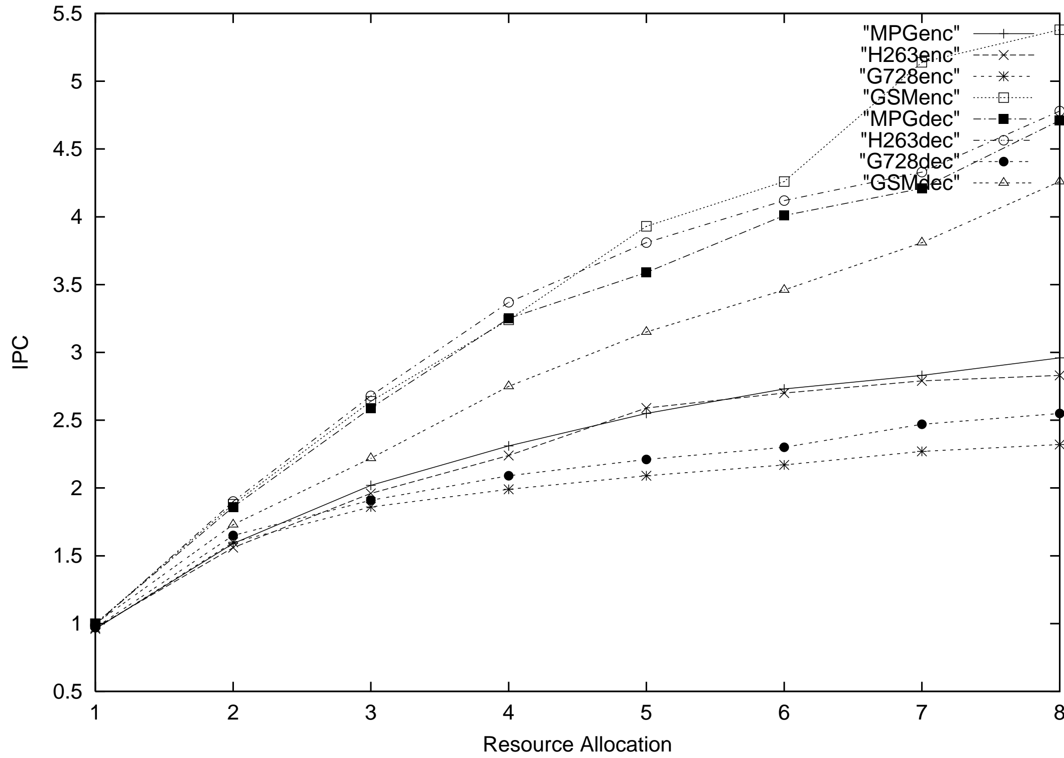

Figure 4 IPC of each multimedia application with different resource allocations. Resource allocation n corresponds to n fetch slots and $16n$ instruction window entries.

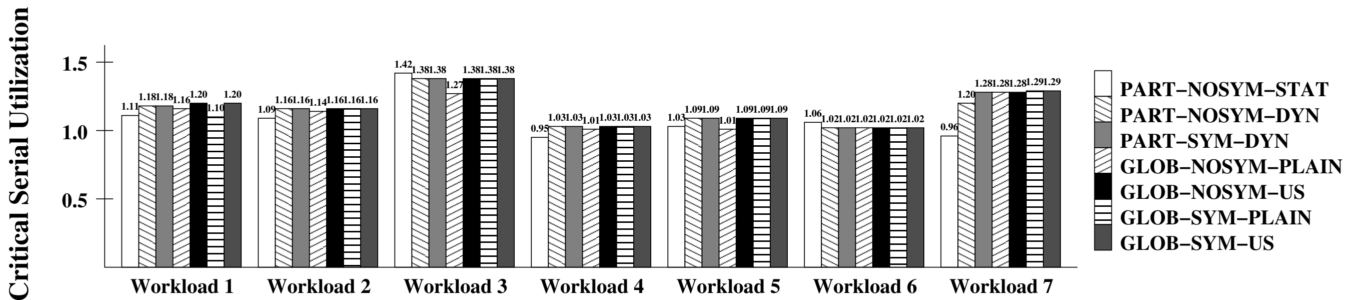

Figure 5 CSUs for the real workloads.

This workload consists of one copy of each decoder. The MPGdec application has a significantly higher utilization than the other applications, making this a workload with skewed utilizations. As expected, the best algorithms are the global scheduling algorithms that give priority to high utilization tasks (GLOB-NOSYM-US and GLOB-SYM-US). They give highest priority to MPGdec, which proves to be the best decision for this workload. PART-NOSYM-DYN constructs the same partitions as PART-SYM-DYN, so symbiosis-awareness has no effect for the partitioning algorithms for this workload. On the other hand, GLOB-NOSYM-PLAIN does better than GLOB-SYM-PLAIN because GLOB-NOSYM-PLAIN gives MPGdec priority behind G728dec and GSMdec, while GLOB-SYM-PLAIN gives MPGdec priority behind G728dec

and H263dec. H263dec has a higher utilization than GSMdec, delaying the scheduling of MPGdec more. Comparing the resource sharing policies, dynamic resource sharing algorithms do better than PART-NOSYM-STAT, which employs static resource sharing. This is expected because as discussed in Section 3, static resource sharing generally compromises throughput for execution time predictability.

Workload 2

This workload consists of one copy of each encoder. The MPGenc and H263enc applications both have much higher utilizations than the other two encoders, with MPGenc's being the highest. The partitioning algorithms place MPGenc in its own partition, guaranteeing that it will execute as a high priority process. The global scheduling algorithms, with the exception of GLOB-NOSYM-PLAIN, also give priority to MPGenc because it is highest utilization and also has the highest symbiosis with the other applications. Therefore, all of the algorithms with dynamic resource sharing do the same, except for GLOB-NOSYM-PLAIN, which does slightly worse. PART-NOSYM-STAT does even worse than GLOB-NOSYM-PLAIN because it has lower throughput due to static resource sharing.

Workload 3

This workload consists of G728 and H263 codecs. The single copy of H263enc has a much higher utilization than the other applications, making this a skewed workload. This workload behaves very similarly to Workload 2, except PART-NOSYM-STAT does the best of all algorithms. Static resource sharing actually performs better on this workload, due to the importance of H263enc. PART-NOSYM-STAT is able to allocate enough processor resources to H263enc such that its individual performance is higher than with dynamic resource sharing, without significantly degrading the overall throughput. Despite the fact that dynamic resource sharing still provides higher total throughput, H263enc misses deadlines more easily for algorithms employing dynamic resource sharing.

Workload 4

This workload consists of GSM and MPG codecs. The single copy of MPGenc has a much higher utilization than the other applications, making this a skewed workload. This workload behaves very similarly to Workload 2, for the same reasons.

Workload 5

This workload consists of G728 and MPG codecs. This workload behaves similarly to Workload 4 because of the presence of MPGenc.

Workload 6

This workload consists of GSM and H263 codecs. The single copy of H263enc has a much higher utilization than the other applications, making this a skewed workload. This workload behaves very similarly

to Workload 3, except GLOB-NOSYM-PLAIN does as well as the other algorithms employing dynamic resource sharing. This is because despite the fact that GLOB-NOSYM-PLAIN assigns H263enc a lower priority than the GSM codecs, GSM codecs do not affect the execution of H263enc much because they have a very small utilization. PART-NOSYM-STAT performs the best because of reasons similar to those for workload 3.

Workload 7

This workload consists of copies of H263dec and MPGdec. These applications have similar utilizations, although MPGdec has a larger utilization. This workload has the least skew in task utilizations, giving the symbiosis-aware algorithms an advantage. Hence, PART-SYM-DYN is the best of the partitioning algorithms, and GLOB-SYM-PLAIN and GLOB-SYM-US are the best global scheduling algorithms, although GLOB-SYM-PLAIN and GLOB-SYM-US do only slightly better than the symbiosis-oblivious global scheduling algorithms.

6 Conclusions

The ability of Simultaneous Multithreaded (SMT) Processors to issue multiple instructions from multiple independent streams of execution every cycle allows them to achieve high throughputs. Soft real-time multimedia applications are an increasingly important workload for a variety of platforms, and have strict computation requirements. While these applications can benefit from the additional throughput of SMT processors, scheduling these applications on SMT processors poses new challenges. There are two levels at which scheduling decisions need to take place. First, we must choose which set of tasks to run simultaneously (determine the co-schedule). Second, we must decide how processor resources should be shared amongst the co-scheduled tasks.

In this work we explore co-schedule selection using both old and new multiprocessor scheduling algorithms, including both partitioning algorithms and global scheduling algorithms. Our new algorithms try to take advantage of the fine-grained resource sharing ability of SMTs (symbiosis-aware algorithms). These algorithms ignore real-time constraints for all but one of the co-scheduled tasks to maximize throughput. We also consider global scheduling algorithms that give priority to high utilization tasks as previously proposed to increase schedulability.

We find that global scheduling algorithms outperform partitioning algorithms due to their ability to migrate tasks from one context to another, which has no associated cost on an SMT. Global scheduling algorithms also have lower algorithmic complexity than most partitioning algorithms, making them an attractive option for SMTs. Symbiosis-aware algorithms are effective, but help most for task-sets consisting of tasks with similar utilizations, where ignoring real-time constraints for most of the co-scheduled tasks is tolerable. Symbiosis-aware algorithms provide little benefit, and can hurt somewhat, for task-sets consisting of tasks with skewed utilizations or with high total utilization. Finally, task-sets consisting of tasks with skewed utilizations see significant benefits from the prioritization of high utilization tasks.

The best overall algorithm is a symbiosis-aware global scheduling algorithm that gives priority to high utilization tasks. This algorithm performs best in almost all cases, but has two major drawbacks, also seen by the other global scheduling algorithms. It does not provide a strict admission control and it can require a lot of profiling for large task-sets or SMTs with a large number of contexts. If these are unacceptable, an alternative algorithm is a partitioning algorithm that utilizes static resource sharing. While its performance is generally lower than the global scheduling algorithms and its algorithmic complexity is somewhat higher, it can provide a strict admission control and requires less profiling than any algorithm employing dynamic resource sharing.

7 Future Work

This study is a first attempt to explore the design space of real-time scheduling algorithms for SMT processors. From our results, we conclude that with the current algorithms, there is a definite trade-off between performance, complexity, profiling overhead incurred, and the admission control provision ability of the algorithm. The PART-NOSYM-STAT algorithm has a low profiling overhead and provides admission control; but at the same time, it has high algorithmic complexity and low schedulability in most cases. The GLOB-SYM-US algorithm, which exhibits the highest schedulability and relatively low algorithmic complexity, has a high profiling overhead and no admission control provision. New algorithms will therefore need to be devised, which try to provide the best of all worlds by having low algorithmic complexity, low profiling overhead, and at the same time, high schedulability and admission control provision.

The main focus of this study has been co-scheduling algorithms. For resource sharing algorithms, we have considered one representative from two classes – the first class tries to maximize overall throughput while the second tries to guarantee a certain level of performance for one or more threads. There is significant scope for more fully exploring the resource sharing design space. For example, the resource sharing algorithm can take advantage of the knowledge of application characteristics in order to bias the resources of the processor in a manner which maximizes throughput. Also, the resource sharing algorithm can dynamically allocate resources to threads based on feedback from the co-scheduling algorithm. It can bias resources in favor of the most critical task (for example – a high utilization task or the earliest deadline task).

References

[1] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, 1990.

[2] Bjorn Andersson and Jan Jonsson. Fixed-Priority Preemptive Multiprocessor Scheduling: To Partition or not to Partition. In *Proc. of the 7th Intl. Conf. on Real-Time Systems and Applications*, 2000.

[3] Garrick Blalock. Microprocessors Outperform DSPs 2:1. *Microprocessor Report*, December 1996.

[4] Bradford L. Chamberlain. Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations. Technical Report UW-CSE-98-10-03, University of Washington, October 1998.

[5] Hao-Hua Chu. *CPU Service Classes: A Soft Real Time Framework for Multimedia Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[6] CNP810SP[tm] Network Services Processor: Key Feature Summary. http://www.clearwaternetworks.com/product_summary_snp8101.html.

[7] Thomas M. Conte et al. Challenges to Combining General-Purpose and Multimedia Processors. *IEEE Computer*, December 1997.

[8] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. In *Operations Research Vol. 26, pp. 127-140*, 1978.

[9] Keith Diefendorff and Pradeep K. Dubey. How Multimedia Workloads Will Change Processor Design. *IEEE Computer*, September 1997.

[10] Christopher J. Hughes et al. Variability in the Execution of Multimedia Applications and Implications for Architecture. In *Proc. of the 28th Annual Intl. Symp. on Comp. Architecture*, 2001.

[11] Christopher J. Hughes, Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors. *IEEE Computer*, February 2002.

[12] Christopher J. Hughes, Jayanth Srinivasan, and Sarita V. Adve. Saving Energy with Architectural and Frequency Adaptations for Multimedia Applications. In *Proc. of the 34th Annual Intl. Symp. on Microarchitecture*, 2001.

[13] Hyperthreading Technology. http://developer.intel.com/technology/hyperthread/.

[14] Stefanos Kaxiras, Girija Narlikar, Alan D. Berenbaum, and Zhigang Hu. Comparing Power Consumption of an SMT and a CMP DSP for Mobile Phone Workloads. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2001.

[15] J. Kreuzinger, A. Schulz, M. Pfeffer, T. Ungerer, U. Brinkschulte, and C. Krakowski. Real-time scheduling on multithreaded processors. In *Proc. of the 7th Intl. Conf. on Real-Time Systems and Applications*, 2000.

[16] S. Lauzac, R. Melhem, and D. Mosse. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *10th Euromicro Workshop on Real Time Systems, pages 188–195*, 1998.

[17] C.L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in A Hard-Real-Time Environment. In *Journal of the Association for Computing Machinery 20(1):46-61*, 1973.

[18] Y. Oh and S. H. Son. Fixed-priority scheduling of periodic tasks on multiprocessor systems. Technical Report 95-16, Department of Computer Science, University of Virginia, March 1995.

[19] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM Reference Manual version 1.0. Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, August 1997.

[20] S. E. Raasch and S. K. Reinhardt. Applications of Thread Prioritization in SMT Processors. In *Proc. of the Workshop on Multithreaded Execution And Compilation*, 1999.

[21] A. Snavely and D. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Architecture. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[22] Anand Srinivasan and Sanjoy Baruah. Deadline-based Scheduling of Periodic Task Systems on Multiprocessors. In *Information Processing Letters*.

[23] Dean Tullsen et al. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of the 23th Annual Intl. Symp. on Comp. Architecture*, 1996.

[24] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parellelism. In *Proc. of the 22th Annual Intl. Symp. on Comp. Architecture*, 1995.