In Proceedings of the 23rd IEEE International Real-Time Systems Symposium, December 2002

Soft Real-Time Scheduling on Simultaneous Multithreaded Processors*

Rohit Jain, Christopher J. Hughes, and Sarita V. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
{rjain2,cjhughes,sadve}@cs.uiuc.edu

Abstract

Simultaneous multithreading (SMT) improves processor throughput by processing instructions from multiple threads each cycle. This is the first work to explore soft real-time scheduling on an SMT processor. Scheduling with SMT requires two decisions: (1) which threads to run simultaneously (the co-schedule), and (2) how to share processor resources among co-scheduled threads. We explore algorithms for both for soft-real time multimedia applications, focusing more on co-schedule selection. We examine previous multiprocessor co-scheduling algorithms, including partitioning and global scheduling. We propose new variations that consider resource sharing and try to utilize SMT more effectively by exploiting application symbiosis*. We find (using simulation) that the best algorithm uses global scheduling, exploits symbiosis, prioritizes high utilization tasks, and uses dynamic resource sharing. This algorithm, however, imposes significant profiling overhead and does not provide admission control. We propose alternatives to overcome these limitations, but at the cost of schedulability.*

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]. The technique of simultaneous multithreading (SMT) [21] seeks to improve resource utilization even further by allowing the processor to fetch, decode, and issue 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 potentially results in higher resource utilization and overall instruction throughput (albeit possibly with some loss of throughput for individual threads). SMT processors are already entering a wide range of markets, including network processors [5] and servers (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 applications. Such applications are becoming increasingly important for general-purpose processors. They often have multiple computationally intensive threads, making them 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 [11]. 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

*This work is supported in part by the National Science Foundation under Grant No. CCR-0096126, EIA-0103645, CCR-0209198, and CCR-0205638, Motorola Inc., and the University of Illinois. Sarita V. Adve is also supported by an Alfred P. Sloan Research Fellowship and Christopher J. Hughes by Intel and Richard T. Cheng graduate fellowships.

place. First, when the number of available tasks[1] is larger than the hardware contexts supported by the SMT processor, we need to determine which tasks to co-schedule (i.e., schedule together). Second, we need to determine how to partition processor resources among 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. These algorithms may make decisions on a cycle by cycle basis, involving fine-grained inter-thread interactions that are generally 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 [20]. Two recent studies considered the co-schedule selection problem [16, 18], but were also throughput based for non-real-time tasks. Two other studies consider SMT resource sharing policies to maximize throughput for a single high priority thread by giving the lower priority threads only leftover resources [7, 17]. One does not consider real-time tasks [7], while the other considers one interactive task (for which it tries to minimize response time) with other non-real-time tasks [17]. Our work concerns periodic real-time applications, where a job is available each period, and any response time up to the end of the task period (or deadline) is acceptable. Finally, there has been some work on real-time scheduling for multithreaded systems where a context switch occurs on a high-latency operation [12]. Since there is no SMT, that work does not have to deal with the resource sharing problem.

This work considers both co-scheduling and resource sharing, but focuses on an exploration of co-scheduling using representative resource sharing policies. We draw on the real-time multiprocessor scheduling literature to consider a variety of co-scheduling approaches, including both partitioning and global scheduling [2, 6, 13, 15]. We also propose new co-scheduling algorithms that consider resource sharing, and seek to utilize the fine-grained resource allocation ability of SMT processors by exploiting *symbiosis* [18] among applications. We evaluate the algorithms using simulation, with synthetic and real workloads.

We find that in terms of schedulability, the best overall algorithm uses global scheduling, exploits symbiosis, gives priority (while co-scheduling) to tasks with high utilization, and uses dynamic resource sharing. This combination allows the algorithm to take advantage of SMT without blindly emphasizing throughput over real-time constraints. Unfortunately, the algorithm has two disadvantages: it does not provide a strict admission control and it requires information on the performance of co-scheduled tasks that may be difficult to acquire. For situations where these limitations are unacceptable, we propose alternatives, but at the cost of lower schedulability.

2 Resource Sharing Algorithms

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 improves processor resource usage and overall throughput, it also means that the performance of each application will 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 throughput in terms of completed instructions per cycle (IPC), ignoring an individual thread's performance (although some employ a fairness criteria). In a real-time system, this could have a negative impact on the schedulability of a task-set. For example, a high utilization task may be given too few resources to make its deadline regardless of when it begins executing. However, providing higher total throughput could have a positive impact on schedulability by allowing some tasks to finish executing earlier than they could without SMT. Thus, we consider two classes of resource sharing algorithms – the first tries to maximize overall throughput while the second tries to guarantee a level of performance for all threads. We experimented with several algorithms; here we report results with one representative algorithm from each class.

Throughput-driven resource sharing – For throughput-driven resource sharing algorithms, we draw on the SMT literature for non-real-time systems [20]. The best reported algorithm, ICOUNT, gives priority to the thread that has the least instructions in the instruction window (i.e., least number of in-flight instructions) [20].

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

The intuition is that the thread with the fewest in-flight instructions is making the most progress; therefore, it is most likely to effectively utilize system resources. We use ICOUNT as the representative throughput-driven resource sharing algorithm in this study. We henceforth refer to it as the **dynamic** algorithm (to contrast with the static algorithm described later). A drawback of the dynamic algorithm for real-time applications is that performance prediction is difficult, because the computation time for a job depends on the behavior of the other co-scheduled jobs and hence on the co-scheduling algorithm. Section 3.2 discusses our solution to this problem.

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 (further discussed in Section 3.2). We consider static resource sharing to represent resource sharing algorithms with performance guarantees. With static algorithms, the basis for allocating a resource reservation to a job is tightly coupled with the co-scheduling algorithm, and is therefore discussed with those algorithms in Section 3.3.

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 involve excessive system overhead and complexity. Previous work found that SMT performance is particularly sensitive to the instruction fetch bandwidth sharing [20]. Experiments with our multimedia application suite and systems showed the same effect. Additionally, our experiments also showed sensitivity to the instruction window allocation. 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. Section 3.2 describes techniques to predict execution time, utilization, and symbiosis for co-scheduled jobs, as required by several of the co-scheduling algorithms. Sections 3.3 and 3.4 describe specific scheduling algorithms based on the identified attributes and prediction techniques. Section 3.5 summarizes the discussion.

3.1 Design Space for Co-Scheduling Algorithms

We classify the design space of SMT co-scheduling algorithms along two axes described below.

Partitioning vs. global scheduling algorithms – The first axis is analogous to that found in the conventional multiprocessor scheduling literature – *partitioning* vs. *global scheduling* [2, 6, 13, 15]. In a partitioning scheme, a partitioning algorithm such as bin-packing is used to bind each task to a processor (a context for SMT), and a uniprocessor scheduling algorithm (e.g., EDF or RM) is used independently on each processor. In a global scheduling scheme, tasks can be executed on any processor and, if preempted, can be resumed on a different processor.

The partitioning approach has received more attention because it provides for admission control and, hence, guaranteed run-time performance. The global approach, on the other hand, was believed to suffer from high overhead (due to frequent task migrations) and anomalous scheduling effects (e.g., the Dhall effect, where task-sets with arbitrarily low utilization have been shown to be not schedulable [6]). However, task migration on an SMT processor is free and recent algorithms have alleviated the schedulability related drawbacks [19]. Recent work has shown significant advantages of global scheduling for soft real-time systems, where worst-case computation time of a task is not representative of the actual time (e.g., [13]). We therefore consider both partitioning and global scheduling based algorithms.

Symbiosis-aware vs. symbiosis-oblivious algorithms – The second axis of classification does not have an analog in conventional multiprocessors. With dynamic resource sharing on an SMT processor, the execution time of a job depends on which other jobs are co-scheduled with it. In particular, $execution\ time = \frac{number\ of\ instructions \times cycle\ time}{instructions\ per\ cycle\ or\ IPC}$, and the IPC of each thread depends on the co-schedule. The best co-scheduling algorithms for an SMT processor may therefore need to consider the impact of co-scheduled tasks on IPC. A measure called the *symbiosis* factor quantifies this impact for N tasks as follows [18]:

$symbiosis\ factor = \sum_{i=1}^{N} \frac{realized\ IPC\ of\ job_i}{single-threaded\ IPC\ of\ job_i}$

The higher the symbiosis factor, the more efficiently is SMT exploited. Ideally, when an SMT processor simultaneously runs N threads, each will have the same performance as when running alone, giving a symbiosis factor

of N. Thus, we distinguish SMT co-scheduling algorithms based on whether or not they consider symbiosis. Figure 1 illustrates the potential benefit of considering symbiosis.

Design space explored – Figure 2 summarizes the design space we explore. We consider both partitioning and global scheduling approaches, using EDF as the underlying algorithm for both. For partitioning, EDF schedules the tasks within a context, and for global scheduling, EDF chooses the next task.

Within each approach, we consider symbiosis-oblivious and symbiosis-aware algorithms. For all cases, we study a dynamic resource sharing policy. We also study a static resource sharing policy for the symbiosis-oblivious partitioning case. We could not identify reasonable algorithms for static resource sharing with the other cases. Static resource sharing primarily provides tight admission control, and we could not exploit this benefit with the other cases.

For partitioning algorithms with dynamic resource sharing, we consider two versions – a simpler base version, and a more complex, but potentially superior, enhanced version, motivated in Section 3.3. For global scheduling, two approaches have been proposed in the past [14, 19], called PLAIN and US here (further discussed in Section 3.4). We explore both, without and with symbiosis.

Of the nine algorithms, the symbiosis-aware ones are particular to SMT. Further, the partitioning algorithms have an additional level of complexity (relative to conventional multiprocessors) since they must account for SMT resource sharing, and possibly allocate resources among partitions.

3.2 Predicting Execution Time, Utilization, and Symbiosis

Several of the co-scheduling algorithms discussed here require knowledge of execution time, utilization, and/or symbiosis of a job when running with other jobs. This section discusses prediction of these quantities.

To predict the execution time (and hence utilization) of a job, we need to predict both the IPC and the number of instructions for the job.

Job IPC in single-threaded mode. Our previous work has shown that on a conventional single-threaded processor, 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) [8]. In this case, therefore, an IPC estimate for all frames can be obtained by profiling one frame of each type at the beginning of the application. On an SMT processor running multiple threads, however, the IPC of a job depends on the underlying resource sharing policy and possibly other co-scheduled jobs.

Job IPC with static resource sharing. Static resource sharing allocates a fixed amount of resources to each task. In this case, 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 obtains the IPC by profiling one job for each task with only the allocated resources in single-threaded mode.

Job IPC with dynamic resource sharing. Dynamic resource sharing makes IPC prediction difficult. For this case, the scheduler profiles all possible co-schedules (i.e., all N-tuples of tasks for an N-context SMT) to obtain the task IPCs. For each co-schedule, we profile for the time it takes to complete the longest frame in the co-schedule, while running successive frames of the other tasks continuously. For the multimedia applications we study, we find that this average IPC also does not change much across different frames of the same application for a given co-schedule.

The above technique makes two assumptions. First, it provides the average IPC at the granularity of the largest frame in the co-schedule rather than that of an individual frame of each application. For most applications considered here, we found this to be reasonable. Second, for an application with multiple frame types, it does not distinguish among the different types.[2]

While the above technique may seem to require a lot of profiling, most multimedia applications run for a large number of frames, making this feasible, if not desirable.

Average job IPC with dynamic resource sharing. For most co-scheduling algorithms, the knowledge of IPCs is required before the co-schedule is determined. When the IPCs depend on the (as yet unknown) co-schedule, an approximation must be made. We use the job IPC averaged across all possible co-schedules as measured above.

Instruction count. The problem of instruction count prediction is not unique to SMT, and techniques developed to predict execution time on conventional uniprocessors would be applicable (e.g., [4]). We use the average instruction count of a large number of frames as the prediction.

Execution time, utilization, and symbiosis. Co-schedule and average execution times and utilizations are directly determined by the instruction count and corresponding IPC estimate. For symbiosis estimates, we additionally require the task IPCs in single-threaded mode, which are easily obtained through profiling.

3.3 Partitioning Algorithms

We extend the notion of partitioning in a multiprocessor system to an SMT processor. For a multiprocessor, a partition is a set of tasks assigned to the same processor. There-

[2] Among the real applications considered here, G728 and MPEG codecs have multiple frame types. For G728, the four different frame types appear consecutively, and so we combine them into one frame, representing systems that buffer four or more frames. For MPEG, we take averages across multiple frames; this works because the dominant frame types (P and B) have IPCs very close to each other.

Task	Period	# instr	IPC with			
			τ_1	τ_2	τ_3	τ_4
τ_1	150	200		4	2	4
τ_2	160	200	4		4	2
τ_3	150	200	2	2		4
τ_4	160	200	4	2	4	

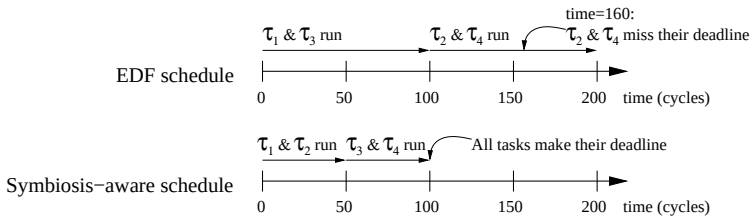

Figure 1. Benefit of symbiosis-aware scheduling. The table on the left shows a task-set with four tasks. For each task, it specifies the period (and deadline), number of instructions, and IPC when co-scheduled with another task on a 2-context SMT processor. The right side shows a simple symbiosis-oblivious EDF schedule and a possible symbiosis-aware schedule on the same processor. The former schedule misses two deadlines while the latter meets all deadlines.

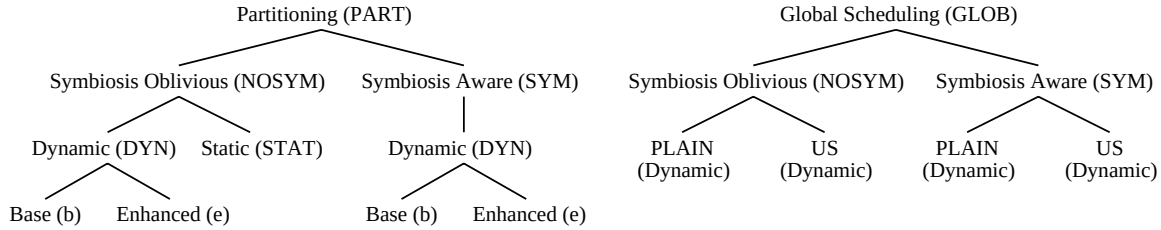

Figure 2. Design space of co-scheduling algorithms. *Dynamic* **and** *static* **refer to resource sharing.**

fore, no two tasks in the same partition can execute simultaneously. Similarly, we consider a partition for an SMT processor to be a set of tasks such that no two will execute simultaneously. Thus, on an SMT processor supporting N contexts, up to N partitions may be created. In a multiprocessor system, partitioning avoids overheads such as the cost of migrating tasks, which is not an issue with SMT. The primary benefit of partitioning with SMT is potentially improved predictability since there are fewer possible interactions among tasks, thereby possibly providing an admission control test. Another difference between multiprocessor and SMT partitioning is the allocation of resources among the contexts. This is fixed for a multiprocessor, but flexible for an SMT. We next describe symbiosis-oblivious and symbiosis-aware SMT partitioning algorithms.

3.3.1 Symbiosis-Oblivious Partitioning

We consider both dynamic and static resource sharing, denoted by PART-NOSYM-DYN and PART-NOSYM-STAT respectively. For the dynamic case, we consider base (b) and enhanced (e) versions.

PART-NOSYM-DYN-b – We start with a bin-packing based algorithm that uses the first-fit-decreasing-utilization (FFDU) heuristic [15]. This sorts the tasks in decreasing order of utilization and allocates a task to the first context that can hold it, as determined by the EDF admission control test (i.e., total utilization in a context should be ≤ 1). The utilizations used here are the averages across all possible co-schedules (Section 3.2). The implementation complexity of this algorithm is lower than the other partitioning algorithms discussed here, but it requires high profiling overhead to determine the average utilizations.

The approximation for utilizations is likely to make the admission test inaccurate for some cases. We therefore consider an enhanced, but more complex, version next.

PART-NOSYM-DYN-e – This algorithm applies two corrections to PART-NOSYM-DYN-b. The first correction modifies the admission test so that no task-set is ever rejected in this phase. When it finds that for a new task, no partition would have a utilization ≤ 1, it increases the threshold utilization by the smallest amount that allows adding the new task to a partition. Since the task utilizations used at this point are approximate, it is possible that the partition is schedulable even with the new task.

The second correction simulates the schedule for a hyperperiod, to determine if it would meet the deadlines. The simulation uses instruction counts and IPCs for the co-schedules as estimated in Section 3.2. If the deadlines are not met, the algorithm moves the smallest utilization task from the violating partition to a partition that meets its tasks' deadlines (using the EDF admission test). A task is never moved into a partition that has ever violated any of its tasks' deadlines (ensuring convergence). This process is repeated until either all partitions become schedulable, or we cannot find a partition to move tasks into, in which case, the task-set is denied admission. The complexity of applying this correction is very high (the profiling overhead is also high). Nevertheless, we study it to give partitioning the fairest showing against global scheduling.

PART-NOSYM-STAT – This algorithm is also based on a bin-packing algorithm using an FFDU heuristic with an EDF admission test. With static resource sharing, the task utilizations are independent of the co-schedule, and so the

corrections in PART-NOSYM-DYN-e are not needed. Instead, the utilizations are dependent on the resource allocation, and this allocation must be determined as part of the co-scheduling algorithm. Our algorithm, which maintains the same resource allocation for the entire run of the task-set, works as follows.

Let $C_1, C_2, \cdots, C_N$ denote the N hardware contexts. At any point in the algorithm, let k be such that no tasks are assigned to context $C_i, i > k$. Initially, all resources are allocated to C_1. The algorithm proceeds such that at any time, contexts $C_2, \cdots, C_k$ are allocated the minimum amount of resources to keep their assigned tasks schedulable.

While assigning a task, if no context $C_j, j \leq k$ can accommodate it, minimal resources are re-allocated from C_1 to C_k such that C_k can accommodate it. If C_1 does not have enough resources, the algorithm fails. The reallocation may cause C_1 to become unschedulable. In that case, we move the smallest utilization tasks from C_1 to the next context that can accommodate 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 reallocation process is repeated between C_1 and C_{k+1}. The algorithm converges because tasks are moved out of, but never returned to, C_1.

The implementation complexity of the algorithm is relatively high because it requires several EDF admission tests for each task assignment and resource reallocation. The profiling overhead for this algorithm, however, is relatively low because only utilizations of tasks running in single-threaded mode are required. The key advantage of this algorithm is that it is similar to a conventional uniprocessor in its ability to provide for admission control (because it does not need any additional approximations for predicting execution time).

3.3.2 Symbiosis-Aware Partitioning

We consider two algorithms for the symbiosis-aware partitioning approach. As mentioned earlier, we only consider dynamic resource sharing for this case.

PART-SYM-DYN-b – This algorithm maximizes average symbiosis among tasks in different partitions, while keeping the total utilization of tasks in each partition 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 its task's utilization. Since dynamic resource sharing is employed, this utilization is approximated as the average utilization across all possible co-schedules. A hypergraph-partitioning algorithm [3] 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 (with up to 20% load imbalance) and the weight of the hyperedges crossing the partitions is minimized (thus maximizing average symbiosis among the different partitions). If a sufficiently load balanced partition is not possible, then the algorithm simply seeks to maximize average symbiosis. The algorithm does not incorporate an admission test (we could, however, incorporate an approximate test by applying the EDF test to all partitions.) The implementation complexity of this algorithm is high (it includes that of the hypergraph partitioning algorithm), and the profiling overhead is also high.

PART-SYM-DYN-e – This corrects for the utilization approximation in PART-SYM-DYN-b by simulating a hyperperiod of the generated schedule, and performing a correction analogous to PART-NOSYM-DYN-e. Its complexity and profiling overhead (and admission control provision) are also analogous to that of PART-NOSYM-DYN-e.

3.4 Global Scheduling Algorithms

3.4.1 Symbiosis-Oblivious Global Scheduling

We evaluate two symbiosis oblivious global scheduling algorithms.

GLOB-NOSYM-PLAIN – This is the EDF (earliest deadline first) algorithm [14]. For an N context SMT processor, the N tasks with the earliest deadlines are chosen. This algorithm is easy to implement and needs no profiling. However, Dhall et al. have shown that task-sets with arbitrarily low utilization can be unschedulable with EDF [6]. Specifically, if one task has a much higher utilization than the others in a task-set (i.e., a skewed distribution), EDF may lead to that task missing its deadline.

GLOB-NOSYM-US – This is the previously proposed EDF-US$[\frac{m}{2m-1}]$ algorithm [19]. It circumvents the Dhall effect [6] by giving the highest priority to high utilization tasks in the task set. The 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 EDF with the following exception. 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). The task utilization is approximated as the average across all co-schedules. The algorithm is implemented as easily as EDF by treating the deadlines of the highest priority tasks as $-\infty$; however, it incurs high profiling overhead to determine the task execution times.

3.4.2 Symbiosis-Aware Global Scheduling

We propose two symbiosis-aware variants of the global scheduling algorithms described above.

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

GLOB-SYM-PLAIN – This algorithm extends EDF to exploit symbiosis in a straightforward way. 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.

Exploiting symbiosis with global scheduling can have both a positive and a negative effect on schedulability. Using co-schedules with high symbiosis improves overall throughput, potentially improving schedulability. However, since the set of threads to run with the EDF-chosen thread is independent of the real-time characteristics of the tasks (e.g., deadline and utilization), this algorithm could potentially reduce schedulability. If the difference in symbiosis for the chosen co-schedule is not large enough to compensate for the lower priority given to the real-time characteristics, this algorithm could hurt schedulability overall.

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 incurs high profiling overhead to determine the symbiosis factors.

GLOB-SYM-US – This algorithm tries to overcome the negative aspects of exploiting symbiosis in GLOB-SYM-PLAIN described above. In particular, in the presence of high utilization tasks (typically, a skewed utilization distribution), prioritizing symbiosis over real-time characteristics is expected to hurt. GLOB-SYM-US, therefore, defaults to GLOB-NOSYM-US if a task T_i has utilization $U_i > \frac{N}{2N-1}$; otherwise, it defaults to GLOB-SYM-PLAIN.

In terms of implementation complexity and profiling overhead, this algorithm is similar to GLOB-SYM-PLAIN. Prioritizing high utilization tasks does not add to complexity, as discussed above.

3.5 Summary

Table 1 summarizes the co-scheduling algorithms studied, with their relative implementation complexity, profiling overhead, and admission control provision.

Algorithm	Implementation Complexity	Profiling Overhead	Admission Control
PART-NOSYM-DYN-b	low	high	approx
PART-NOSYM-DYN-e	highest	high	approx
PART-NOSYM-STAT	high	low	yes
PART-SYM-DYN-b	high	high	no
PART-SYM-DYN-e	highest	high	approx
GLOB-NOSYM-PLAIN	lowest	none	no
GLOB-NOSYM-US	lowest	high	no
GLOB-SYM-PLAIN	low	high	no
GLOB-SYM-US	low	high	no

Table 1. Implementation complexity, profiling overhead, and admission control provision (for which "yes" means admission control similar to that on a conventional uniprocessor, and "approx" means some admission control, but less strict than "yes").

4 Experimental Methodology

We evaluate the various scheduling algorithms with simulation of an SMT processor with two hardware contexts. We first evaluate with synthetic workloads to cover a large workload space (Section 4.1), and then use a cycle-accurate simulation of some real workloads (Section 4.2).

4.1 Synthetic Workload Evaluation

4.1.1 Synthetic Workload Description

The synthetic workloads are generated using a methodology similar to that in [2]. We consider separate workloads to represent cases where the constituent tasks have utilizations following a normal or a bimodal (i.e., skewed) distribution. Both cases occur for multimedia workloads in practice (e.g., video encoders have much higher utilization than speech codecs or video decoders, as in Table 2). For the normal distribution case, we consider five values for the expected mean utilizations, to give a total of six workloads. For each workload, we report results averaged over 2,000,000 task-sets, randomly generated as follows.

For all workloads, the number of tasks in a task-set, 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 the set $\{100, 200, 300, 400, 500, \cdots, 1600\}$, with uniform probability. For the normal distribution cases, the mean 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 $0.5E[u_i]$. If $u_i < 0$ or $u_i > 1$, a new u_i is generated. We evaluate for $E[u_i] = 0.15, 0.20, 0.25, 0.30$, and 0.35. For the bimodal distribution case, in each task-set, the utilization of one task, τ_i, follows a normal distribution with an expected value of $E[u_i] = 0.9$ and standard deviation of 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, in the task-set, follows a normal distribution with an expected value of $E[u_j] = 0.03$ and standard deviation of 0.01. If $u_j \leq 0$ or $u_j > .1$, a new u_j is generated.

For all workloads, the 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 1.0. If $IPC_i < 0$ or $IPC_i > 6$, a new IPC_i is generated. The mean 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). $Stddev[I_i]$ 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.

We model a dynamic resource sharing policy for the synthetic workloads. Each task has a set of IPCs: one for when it is standalone (described above), and one each for when it is co-scheduled with the other tasks ($IPC_{i,j}$ for task pair (τ_i, τ_j)), generated as follows. For task pair (τ_i, τ_j), the reduction in the standalone IPC of task τ_i 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 $IPC_{i,j} \leq 0$ or $IPC_{i,j} > IPC_i$, a new $IPC_{i,j}$ is generated.

For synthetic workloads, we do not evaluate PART-NOSYM-STAT because modeling static resource sharing requires a different way of generating IPCs for tasks, precluding a fair comparison with the other algorithms.

4.1.2 Metrics

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, over the simulation of a hyperperiod. Since we consider soft real-time systems, a task-set is considered successfully scheduled even if some deadlines are missed. We allow 5% of the deadlines to be missed for each application.

We report the success-ratios for each workload as an aggregate (i.e., across all 2,000,000 task-sets generated for the workload). For deeper analysis, we also divide each workload into bins, based on the average total utilization of the task-sets, and collect success-ratios for each such bin. The average total utilization for a task-set is computed as the sum of the average utilizations of all tasks (averaged across all co-schedules, as in Section 3.2). We consider bins with average total utilization intervals $[0.00-0.05), [0.05-0.10)...[1.95-2.00)$. Task-sets with total utilization ≥ 2.0 are not considered because they are unlikely to be schedulable by any algorithm (given a two context SMT); however, these are included in the aggregate workload results.

4.2 Real Workload Evaluation

Our real workloads draw from eight video and speech encoder and decoder applications, summarized in Table 2 and described in [8]. The specific combinations of applications studied are discussed along with the results in Section 5.2.

We use the detailed, cycle-level RSIM architecture simulator [9] to model a uniprocessor system with an SMT processor running the real workloads for one hyperperiod. The processor modeled is a straightforward extension of a conventional out-of-order superscalar design [20] and is summarized in Table 3. It can simultaneously execute up to two threads. This study assumes a perfect instruction cache.

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 2. Real applications. The IPC and utilizations are when running alone on the simulated system in Table 3. Symbiosis factors are reported in [10].

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

Table 3. Parameters for the base architecture for real workloads.

For the real workloads, we use a metric called *critical serial utilization*. This is based on the critical utilization for conventional processors, which is defined as the total utilization obtained by uniformly increasing the utilization of all tasks until a further increase causes the task-set to become unschedulable. As before, we consider a task-set as unschedulable by an algorithm if the fraction of deadlines missed for any task is $> 5\%$.

For an SMT processor, we cannot directly use critical utilization because a job's execution time depends on the co-schedule. We instead define the *serial utilization*, S, for an SMT processor to be analogous to the utilization of a conventional processor. For a task set $\tau_1, \tau_2, \cdots, \tau_n$ on an SMT processor, $S = \sum \frac{C_i}{P_i}$, 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 and a scheduling algorithm) as the total serial utilization obtained by uniformly increasing the utilization of

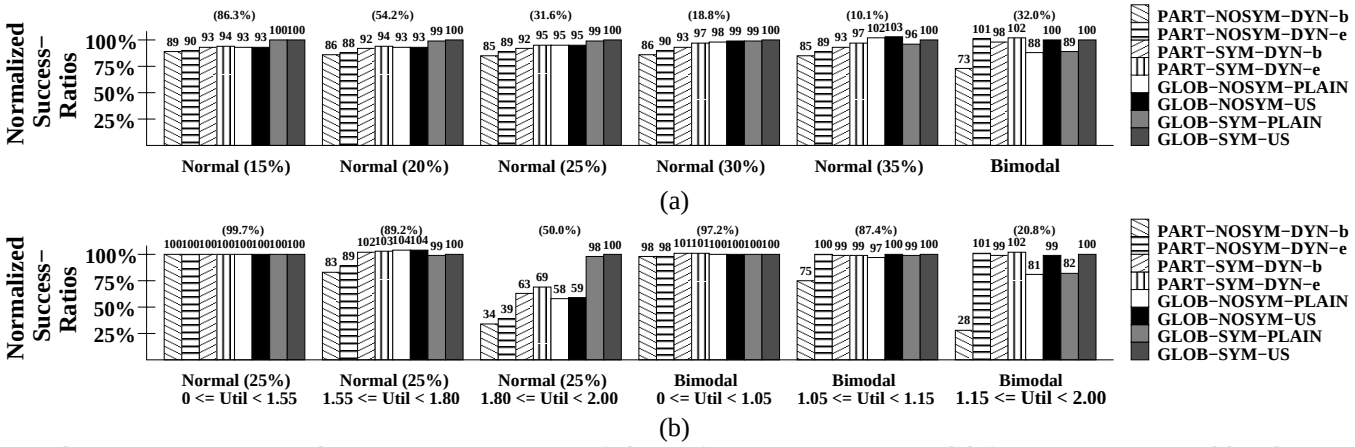

Figure 3. Success-ratios, as a percentage of those for GLOB-SYM-US, (a) for aggregate workloads, and (b) for three total utilization bins for two workloads (normal with mean 25% and bimodal). The absolute numbers for GLOB-SYM-US are in parentheses above each workload and bin.

the tasks in the set until a further increase in the utilization of any of the tasks causes the task-set to become unschedulable. It is possible for an SMT processor to achieve a CSU > 1 since the tasks can overlap with each other. To measure the CSU for a workload and scheduling algorithm, we reduce the periods of all the applications in the workload by the same factor until the workload becomes unschedulable.

The CSU is analogous to the success ratio used for the synthetic workloads. An algorithm with a higher success ratio is expected to have a higher average CSU, which allows a comparison between synthetic and real workload results.

5 Results

Sections 5.1 and 5.2 present the results of our evaluation with the synthetic and real workloads respectively.

5.1 Evaluation of Synthetic Workloads

For the workloads discussed in Section 4.1, Figure 3(a) shows the success-ratio of eight scheduling algorithms, normalized to that of GLOB-SYM-US. For further analysis, Figure 3(b) shows more details for two representative workloads – normal distribution with 25% mean and bimodal distribution. As discussed in Section 4.1, we divide each workload into forty bins based on total utilization. Figure 3(b) aggregates these bins into three, based on trends in success-ratios, and reports the average success-ratio of each (normalized to GLOB-SYM-US). We refer to the three bins for each workload as the low, moderate, and high utilization bins, but note that the absolute values of utilizations covered by each vary with workload.

Best algorithm – In terms of schedulability, GLOB-SYM-US is the best or within 3% of the best algorithm across all workloads. Considering the bins, all algorithms give $> 95\%$ absolute success-ratios at low total utilizations for all workloads (up to 1.45 utilization for the normal distributions and 1.05 for bimodal). However, for higher utilizations (the target for an SMT), the algorithms show significant differences. In particular, every algorithm is 18% or more worse than GLOB-SYM-US for the high total utilization bin of at least one of the workloads. GLOB-SYM-US, on the other hand, is the best or within 5% of the best algorithm, even when considering the finer bins. The high overall performance of GLOB-SYM-US across all workloads and bins shows it is the most successful at combining the best algorithmic attributes discussed in Section 3.1.

In terms of algorithmic complexity and profiling overhead, the most attractive algorithm is GLOB-NOSYM-PLAIN (see Table 1); however, its advantages come at the cost of lower schedulability overall than GLOB-SYM-US. All other approaches are either comparable or higher in complexity/overhead and/or significantly poorer in success-ratios, compared to GLOB-SYM-US. Thus, even in terms of algorithmic complexity and profiling overhead, this algorithm provides a good tradeoff. The rest of this section evaluates individual attributes for the various algorithms.

Partitioning vs. global algorithms – The simplest partitioning algorithm, PART-NOSYM-DYN-b, is inferior to all the global scheduling algorithms for all workloads. Although the task utilizations used for scheduling are approximate, PART-NOSYM-DYN-b uses them in a rigid way and precludes any runtime migration of tasks. As total utilization of the task-set increases, the above limitation results in significant load imbalance and a large reduction in performance.

The addition of symbiosis-awareness and/or the enhancements discussed in Section 3.3.1 alleviates the limitation and makes partitioning more competitive with global scheduling. In some cases, some of the partitioning approaches are even superior to some of the global algorithms

(but not to GLOB-SYM-US by any significant amount). Specifically, with the bimodal workloads, the enhanced partitioning approaches are the best overall. They assign the high utilization task to its own partition, allowing it to execute all the time and improving schedulability to be slightly higher than even GLOB-SYM-US. These algorithms, however, are more complex than global scheduling.

Symbiosis-awareness – Considering symbiosis helps in most cases, although the benefits for the aggregate workloads are modest (maximum of 9%, with the exception of one case for PART-NOSYM-DYN-b). The individual bins, however, show that symbiosis awareness is critical in some cases, particularly for workloads with normally distributed utilizations (more than 68% benefit for all algorithms for the high utilization bin with 25% mean).

The aggregate normal distribution workload with 35% mean, however, sees a degradation with symbiosis-aware global scheduling. Figure 3(b) shows a similar degradation for the moderate utilization bin of the normal 25% mean workload. As discussed in Section 3.4.2, when the symbiosis increase with the chosen co-schedule is insufficient to compensate for the negative impact of ignoring real-time characteristics, symbiosis-aware global algorithms can hurt performance. GLOB-SYM-US overcomes part of this limitation (by defaulting to GLOB-NOSYM-US with a high utilization task), and sees slightly lower overall degradation than GLOB-SYM-PLAIN.

Our detailed results show that for every normal distribution workload, there is a range of total task utilizations (the moderate bin in Figure 3(b)) where symbiosis-awareness hurts the global algorithms. Below this range, all algorithms perform well because the task utilizations are low enough to be easily schedulable. Above this range, most algorithms have relatively low success-ratios. Symbiosis-aware algorithms perform the best in this (high) range because they reduce task utilizations, making more task-sets schedulable that would otherwise have failed. Depending on the actual success-ratios and the number of task-sets in the moderate and high bins, either the GLOB-SYM-* or GLOB-NOSYM-* version is best for the aggregate workload. With increasing mean utilization, it becomes harder even for symbiosis-aware algorithms to schedule many tasks in the high bin; therefore, with mean utilization of 35%, symbiosis-awareness hurts slightly overall. (The success-ratio for this workload, however, is only 10%.)

Based on the above, a heuristic where GLOB-SYM-* defaults to GLOB-NOSYM-* depending on the mean and/or the total utilization could make symbiosis-aware algorithms even more successful. We leave this to future work.

For the bimodal workload, the symbiosis-aware global algorithms are similar to the corresponding symbiosis-oblivious ones. Symbiosis-aware US defaults to ignoring symbiosis. The SYM and NOSYM versions of PLAIN both perform almost the same, but are significantly worse than the US versions. Thus, in the bimodal case, prioritizing the high utilization task is the more important distinguishing effect than whether or not symbiosis is exploited.

Finally, considering symbiosis is always helpful for (or does not appreciably hurt) partitioning algorithms. These algorithms use symbiosis only to determine a partitioning; within each partition, EDF respects real-time deadlines.

Enhancements of partitioning algorithms – PART-NOSYM-DYN-e always improves upon PART-NOSYM-DYN-b, and by a large amount in some cases. Our detailed results show that most of this improvement comes from the heuristic of allowing total utilization of a partition to exceed 1 while initially partitioning. The more expensive hyperperiod simulation contributes less to the overall performance. Similarly, PART-SYM-DYN-e also always improves upon PART-SYM-DYN-b, but by a relatively modest amount.

PLAIN vs. US algorithms for global scheduling – Comparing GLOB-NOSYM-PLAIN vs. GLOB-NOSYM-US and GLOB-SYM-PLAIN vs. GLOB-SYM-US, we see that there is little difference between PLAIN and US for the normal distribution case. This is because there is little skew in the task utilizations, making US effectively function as PLAIN. Task-sets with larger mean task utilization are likely to have more skew, and we see the US algorithm do very slightly better than the corresponding PLAIN algorithm for mean task utilization of at least 30%. For the bimodal distribution, the US version performs 12% to 14% better than the corresponding PLAIN version. As expected, this comes from giving priority to high utilization tasks.

5.2 Evaluation of Real Workloads

We now evaluate workloads based on the real applications in Section 4.2. We evaluated several combinations of the applications to simulate teleconferencing workloads. For lack of space, we only present representative workloads that show differences among the various algorithms. Our goal here is to show that the space covered by the synthetic workload evaluation is representative of real workloads, and the conclusions drawn apply.

Figure 4 shows the CSUs for five workloads (normalized to those for GLOB-SYM-US). The encoders and decoders comprising each workload are given below the corresponding CSU bars. For workload 2, G728' is a version where 20 frames of the G728 application described in Table 2 are buffered and processed together (as is often the case with speech applications), giving a longer period of 50ms.

We find that the conclusions from the synthetic workloads still hold, although the absolute differences among the algorithms are generally smaller.

Best algorithm – GLOB-SYM-US again is the best algorithm overall – it always performs better than or very

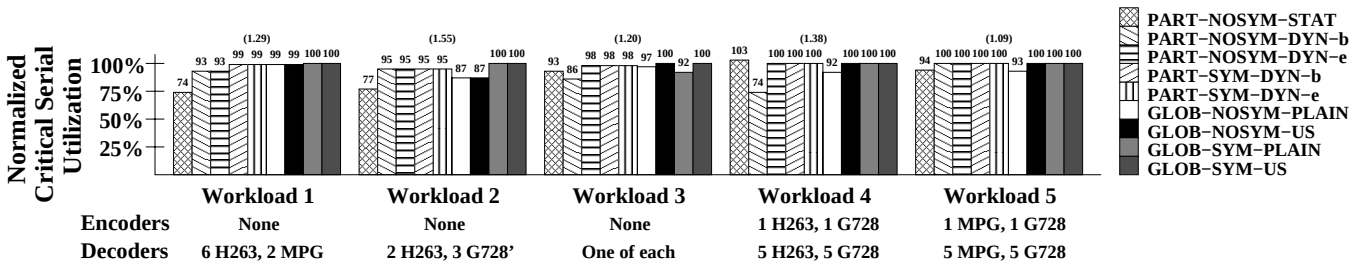

Figure 4. CSUs for real workloads, normalized to CSU for GLOB-SYM-US (in parentheses).

close to all the other algorithms. Compared to GLOB-NOSYM-PLAIN, which is the least complex of the algorithms employing dynamic resource sharing, GLOB-SYM-US is better in almost all cases (maximum difference is 13%). These results also include PART-NOSYM-STAT, which employs static resource sharing, has low profiling overhead, and provides for admission control. Overall, GLOB-SYM-US is superior to PART-NOSYM-STAT (maximum benefit is 26%); however, in one case, PART-NOSYM-STAT is marginally better (by 3%).

We next discuss the impact of the individual design attributes, thereby also explaining the above results. At the point where the CSU is measured, workloads 1 and 2 in Figure 4 have utilizations analogous to the synthetic workloads with normal distributions, while workloads 3, 4, and 5 are analogous to the bimodal workloads (with MPGdec, H263enc, and MPGenc respectively having high utilizations). We therefore refer to them as normal or bimodal respectively below.

Static vs. dynamic resource sharing – Static resource sharing generally implies lower throughput than dynamic resource sharing. This results in poor CSU for PART-NOSYM-STAT for the two normal workloads. However, with the bimodal workloads, PART-NOSYM-STAT is more competitive and even surpasses GLOB-SYM-US in one case (albeit by only 3%). With the bimodal workloads, static resource sharing dedicates sufficient resources to the high utilization application, enabling it to meet its deadlines better than with dynamic resource sharing. This effect mitigates the impact of the degraded throughput due to a static policy. The rest of this section focuses on the algorithms with dynamic resource sharing.

Partitioning vs. global algorithms – As in the synthetic case, the simplest partitioning algorithm, PART-NOSYM-DYN-b, is not competitive in general. The advanced partitioning algorithms (with dynamic resource sharing) are more competitive and very close to GLOB-SYM-US in most cases, but with higher complexity.

Symbiosis-awareness – For partitioning, symbiosis-awareness again never hurts and often helps (maximum benefit of 35% for the DYN-b version and a modest 8% for the DYN-e version). For global scheduling, symbiosis-awareness often helps (maximum benefit of 15% for both PLAIN and US versions); however, for workload 3, it hurts the PLAIN version by 5%. Here, MPGdec and H263dec both have the same (high) symbiosis factor with G728dec (1.65). Breaking the tie arbitrarily, GLOB-SYM-PLAIN chooses H263dec as the application to co-schedule with G728dec. This is a poor choice because MPGdec has a higher utilization and shorter period. (GLOB-SYM-US does not suffer from this because it defaults to GLOB-NOSYM-US for bimodal workloads).

Enhancements for partitioning – The enhancements provide significant benefits for the NOSYM version in two cases – 14% and 35% (workloads 3 and 4 respectively). Our detailed results show that the former comes mainly from hyperperiod simulation while the latter comes mainly from threshold enhancement. The SYM version does not see any benefits from the enhancements.

PLAIN vs. US for global scheduling – The US version benefits the bimodal workloads (maximum of 9% for both NOSYM and SYM versions). Bimodal workloads 4 and 5 do not see any benefit from US for the GLOB-SYM version because coincidentally, the application with the highest symbiosis factors is also the one with the highest utilization (H263enc and MPGenc respectively), resulting in the same schedules for the SYM-PLAIN and SYM-US versions.

6 Conclusions

To our knowledge, this is the first study on soft real-time scheduling for simultaneous multithreaded (SMT) processors. There are two aspects to the scheduling problem with SMT. First, we must choose a set of tasks to run simultaneously (i.e., determine the co-schedule). Second, we must decide how to share processor resources among co-scheduled tasks. We study both aspects, but focus on co-scheduling algorithms. We explore previous multiprocessor scheduling approaches, including partitioning and global scheduling; for global scheduling, we also consider versions that prioritize high utilization tasks. For the partitioning approaches, we show how to account for dynamic resource sharing in an SMT processor and also provide an algorithm to statically allocate resources. In all cases, we propose new variants that try to exploit the fine-grained resource sharing ability of SMT processors by giving preference to applica-

tions that maximize co-schedule symbiosis over those with potentially tighter real-time constraints.

We evaluate our algorithms on a two-context SMT processor, using synthetic and real multimedia workloads. We find that in terms of schedulability, the best algorithm uses global scheduling, exploits symbiosis, prioritizes high utilization tasks, and uses dynamic resource sharing. This algorithm has two drawbacks: it does not provide a strict admission control, and it can require a lot of profiling for large task-sets or SMT processors with a large number of contexts. If these are unacceptable, an alternative is a partitioning algorithm that utilizes static resource sharing. While it performs worse overall in terms of schedulability and is somewhat more complex, it can provide a strict admission control and requires less profiling. Another alternative is the earliest deadline first global algorithm, which generally performs worse than the best algorithm and does not provide strict admission control, but requires no profiling.

Considering individual design decisions, we find that generally dynamic resource sharing is better than static for schedulability. Partitioning algorithms can be made to be competitive with global scheduling algorithms, but with more complexity. Symbiosis-awareness is beneficial for partitioning algorithms because they do not entirely ignore real-time constraints. Symbiosis-awareness can hurt or help global scheduling algorithms, depending on the relative magnitude of the symbiosis factors and total utilization of the applications. Symbiosis also does not help in the case of applications with skewed utilizations. Alternate heuristics could be found where symbiosis is considered only for the right combination of utilization characteristics and symbiosis factors; our GLOB-SYM-US algorithm is a step in that direction, and we leave further exploration to future work. Finally, we find that task-sets consisting of tasks with high utilizations see significant benefits from the prioritization of those tasks.

In the future, we plan to evaluate our algorithms with more than two SMT contexts, develop improved heuristics for our algorithms as motivated by this work, and consider scheduling of parallel applications in conjunction with other independent applications on an SMT.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proc. of the 1990 Intl. Conf. on Supercomputing*, 1990.

[2] B. Andersson and J. 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] B. L. Chamberlain. Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations. Technical Report UW-CSE-98-10-03, Univ. of Washington, Oct. 1998.

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

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

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

[7] G. K. Dorai and D. Yeung. Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance. In *Proceedings of the 11th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2002.

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

[9] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors. *IEEE Computer*, February 2002.

[10] R. Jain, C. J. Hughes, and S. V. Adve. Supplemental Data for "Soft Real-Time Scheduling on Simultaneous Multithreaded Processors". URL: http://www.cs.uiuc.edu/rsim/Pubs/rtss02-supplement.ps.

[11] S. Kaxiras, G. Narlikar, A. D. Berenbaum, and Z. Hu. Comparing Power Consumption of an SMT and a CMP DSP for Mobile Phone Workloads. In *Proc. of the Intl. Conf. on Compilers, Arch. and Synthesis for Embedded Systems*, 2001.

[12] 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.

[13] 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*, 1998.

[14] C. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in A Hard-Real-Time Environment. In *Journal of the ACM 20(1):46-61*, 1973.

[15] Y. Oh and S. H. Son. Fixed-Priority Scheduling of Periodic Tasks on Multiprocessor Systems. Technical Report 95-16, Dept. of Comp. Science, Univ. of Virginia, March 1995.

[16] S. Parekh, S. Eggers, and H. Levy. Thread-Sensitive Scheduling for SMT Processors. Technical Report, Dept. of Comp. Science and Eng., Univ. of Washington, 2000.

[17] 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.

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

[19] A. Srinivasan and S. Baruah. Deadline-based Scheduling of Periodic Task Systems on Multiprocessors. *Information Processing Letters*, October 2002.

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

[21] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parellelism. In *Proc. of the 22th Intl. Symp. on Computer Architecture*, 1995.