Asymmetric Multiprocessing for Simultaneous Multithreading Processors *

Daniel M. Smith, Vincent W. Freeh, Frank Mueller
Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7534
{dmsmith2,vwfreeh,fmuelle}@ncsu.edu

Abstract

Simultaneous Multithreading (SMT) has become common in commercially available processors with hardware support for dual contexts of execution. However, performance of SMT systems has been disappointing for many applications. Consequently, many SMT systems are operated in a single-context configuration to achieve better average throughput, depending on the application domain.

This paper first examines the performance of several benchmarks programs in both uniprocessor mode (one hardware thread context) and SMT mode (two contexts) on a Hyperthreaded Pentium 4. Based these results, three classes of applications can be distinguished: 1) those that prefer to execute in SMT, 2) those that prefer uniprocessor-mode, and 3) those that do not care. Our classification takes into account two aspects: (a) the performance of the application itself *when sharing the system and (b) the performance of* other applications *whose concurrent execution is being affected.*

The paper introduces a novel bi-modal *scheduler for Linux that continuously alternates between two distinct modes:* SMT *and* UNI*. The SMT mode uses the standard Linux SMP scheduler with its support for SMT while the UNI mode uses the standard uniprocessor scheduler, activating only one hardware context. Thus, the system alternates between operating as an SMT and operating as a uniprocessor even though its hardware is multithreaded. An application labeled as UNI will only execute in the UNI mode phase while SMT applications run concurrently. In addition, we provide a third class, called* ANY*, for processes that are agnostic with regard to SMT.*

Need a results sentence here!

1. Introduction

Simultaneous Multithreading (SMT) is a processor design that supports multiple threads with separate hardware contexts to dispatch instructions in parallel on a single processor core, *i.e.*, SMT creates multiple *logical* processors within a single physical processor. In contrast, an SMP (or a CMP—chip multiprocessor [10, 15]) executes concurrent threads on distinct physical processors in which sharing starts in the memory hierarchy, usually at the L2 or the L3 cache.

SMT was motivated by the limitations on instruction-level parallelism within single-threaded applications. Hence, SMT replicates some microprocessor state (*e.g.,* register files) for each thread context it can concurrently execute, but the majority of microprocessor resources are shared, including functional units and the L1 cache. A few shared resources are partitioned equally to all threads (*e.g.,* queues) but most resources are allocated dynamically on demand (*e.g.,* caches). Because concurrent threads compete for more resources in an SMT, there is greater potential for interference and performance degradation than in an SMP. However, there are potential benefits to this increased sharing. In particular, sharing all of the cache hierarchy can make inter-thread communication and synchronization inexpensive.

An SMT microprocessor fetches, issues, and executes instructions from multiple threads on every cycle. It can process instructions from different threads without hardware or software context switches. This is a distinct advantage over single-threaded uniprocessors or traditional multithreaded microprocessors [5, 6]. The resultant interleaving of streams has two primary advantages. First, an SMT can more easily tolerate memory latency because it can execute instructions from another thread. Second, concurrently executing multiple threads tends to increase the utilization of the microprocessor due to better instruction mix. These two advantages are referred to as reducing *vertical* and *horizontal* waste, respectively [24].

SMT has a small additional cost but large potential gains. This is very different from other architecture features, such as super-scalar execution and branch prediction, in which performance gains (up to five-fold) are less than the additional resources committed (a 15- to 18-fold increase in resources) [12]. In contrast, In-

* This work was supported in part by NSF grant CNS-0410203.

tel reports that a Pentium 4 with Hyper-Threading Technology (Intel's SMT implementation) shows a 30% increase in performance with only a 5% increase in die space [12]. Not surprisingly, SMTs are being adopted in many contemporary architectures, ranging from processors by Intel to IBM and Sun.

Overall, SMT is a novel microarchitecture that has the potential to boost performance significantly. However, performance of SMT systems has been disappointing for many applications. While peak performance gains of up to 30% have been observed for dual contexts [12], SMT is reported to adversely affect the performance of many applications in practice. In fact, many SMT systems today are operated in a single-context configuration to achieve better average throughput, depending on the application domain.

This difference in application behavior can, in part, be attributed to a lack of operating system support for SMTs. In a naïve approach, an SMT is treated as a symmetric multiprocessor, where each *logical* processor in the SMT is treated as a fully-vested physical processor. While this approach enables basic SMT functionality, it does not fully exploit the potential benefits of SMT. Just as the introduction of multiprocessors necessitated innovation, the arrival of commercially available SMT microprocessors calls for new software architecture. Specifically, there is a need to develop new mechanisms to expose and control SMT features. Further, there is a need to develop new execution models for operating systems, applications and middleware that exploit SMT.

In this paper, we assess the short-comings of pure hardware support to SMT and, instead, promote a synergistic approach between the operating system and SMT hardware with two logical processor contexts. We show that event-driven activities and kernel tasks can almost entirely be directed to one logical processor to spare the second one from interrupts or other kernel code. We term this an Asymmetric Multiprocessing (AMP) mode. A dedicated kernel context effectively transforms a conventional dual-context SMT system into one that closely resembles the performance characteristics of a single-context system while handling kernel code on the second context. More significantly, we show that applications can easily be characterized into two classes: one that benefits from SMT systems and another that thrives in a single context AMP configuration when kernel activity is present. Our hybrid design exploits such characterization by dynamically switching between one and two contexts, depending on the active application. We further demonstrate benefits of AMP in increased responsiveness and predictability of application performance in the presence of high interrupt arrival rates. Hence, our synergistic OS/hardware approach is not only beneficial for alternating executions of applications with different SMT characteristics, it also provides sustained responsiveness and predictability during unexpected and frequent interrupts.

The paper is structured as follows. Sections 2 and 3 present the design and implementation of our AMP mode, respectively. Sections 4 and 5 provide the framework and results from experiments, respectively. Section 6 relates our contributions to prior work. Section 7 summarizes our work.

2. Asymmetric-Multiprocessing Scheduler

In this section, we describe the design of a scheduler for asymmetric multiprocessing (AMP). Our work is originally motivated by a lack of system support for exploiting SMT architectures, and our design aims at integration into the Linux kernel. Nonetheless, the concepts of AMP generalize beyond Linux and are extensible beyond dual-context SMT architectures to a larger number of contexts.

2.1. Scheduler Design

AMP is supported by modifications to a conventional operating system scheduler. We integrated AMP into the Linux 2.6 kernel as a proof-of-concept for our design. In the following, we will briefly outline the design of the related kernel structures and algorithms. Our changes to Linux are made primarily in the actual kernel source code. Other additions are mostly support-related (*e.g.*, the control interface) and are encapsulated within a loadable kernel module.

The original Linux task scheduler is effectively unbiased between kernel and user tasks. Hence, tasks can be scheduled on either context of an SMT. Our AMP design, in contrast, ensures that kernel tasks and user tasks are assigned to separate contexts. As such, the processor can alternate between an unpartitioned mode favoring user tasks and a partitioned mode where kernel tasks and user tasks execute in separate contexts.

Tasks are assigned to processor contexts in AMP mode by the algorithm depicted in Figure 1. Note that this design builds on the existing algorithm present in Linux and only requires minor modifications. The first conditional block is an existing mechanism for providing CPU affinity in an SMP system. The `allowed_cpus` field of the task structure is a bit mask, where a set n-th bit indicates that processor n can execute the task. The second conditional block is novel and unique to our design. It checks if the next task selected for execution should ex-

```
schedule_next_task() {
BEGIN:
  /* Existing performance tweak */
  if (!match(next→allowed_cpus, this_cpu)) {
    next = get_next_task();
    goto BEGIN
  }

  /* New assignment code */
  if (match(task→type, this_cpu_duty)) {
    goto RUN_TASK;
  } else {
    next→allowed_cpus -= this_cpu;
    next = get_next_task();
    goto BEGIN
  }
RUN_TASK:
}
```

Figure 1. Modified scheduling algorithm for AMP mode

ecute on the current context according to the AMP policy.

In practice, the new code is only executed a small number of times while the system adjusts the task list appropriately. As tasks transition from a blocked to runnable state in AMP mode, they are marked for execution only on the correct context. Thus, after every process in the system has been examined, each is marked to run on the correct context. Since both contexts fetch from the same run queue, tasks not allowed to execute on a context are quickly identified at the top of the routine without re-executing the logic detailed in Figure 1. This modification only slightly changes the existing task selection logic and, hence, can be selectively enabled or disabled at runtime. Thereby, the operating system (or user) can select the mode that provides the best performance.

The AMP mode also impacts the idle state during scheduling. In Linux, a special kernel task, the idle task, is scheduled on unused processors when the number of processors exceeds the number of runnable processes. On an SMT, the idle task explicitly issues a halt instruction, which delays execution until the next hardware interrupt. When issuing halt on one context of an SMT, all shared resources are relinquished to the other running context on the Intel architecture, *i.e.*, the processor effectively becomes unpartitioned. Hence, when there is only one runnable task, all processor resources are allocated to that task. The processor is effectively a uniprocessor, which allows the running task potentially to increase its performance. When operating in AMP mode, the context dedicated to kernel tasks rejects any user tasks so that the idle task is invoked in the absence of runnable kernel tasks. Hence, resources are relinquished in favor of user tasks whenever possible, *i.e.*, when no kernel tasks are pending.

2.2. A Context for Event Handling

The kernel code of an operating system is comprised of integer-based work. But in contrast to integer-based applications, kernel code performs worse. Previous studies have shown that the operating system executes so infrequently that branch predictions and cache performance suffer [17, 9, 4]. This behavior results in favorable conditions for user applications co-scheduled in the adjacent context during AMP mode. First, the operating system is interleaved with application execution. Hence, poor performance in kernel routines will not significantly delay the execution of the application. Second, the dynamic nature of resource binding in Intel SMTs causes the poorly-performing kernel code to utilize few resources, thereby minimizing the impact of the concurrently executing user task.

Our subsequent experiments will show that benefits can be found when dissimilar resources are utilized. For example, floating-point intensive computations introduce constraints that can be exploited by AMP. We also observed benefits for integer-based applications. Overall, some floating-point and some integer applications benefit while others do not.

The AMP mode provides a clear separation between kernel and user tasks. This design is particularly suitable for interrupt handling, such that kernel tasks due to events are handled on a dedicated context. In traditional systems, frequent interrupts impede the progress of user tasks and can adversely affect their performance due to pollution of caches, branch predictors and branch target buffers (BTBs) as well as other resources. The AMP mode, in contrast, provides a natural boundary between user tasks and interrupt activity by separating them into different hardware contexts. Certain resources are replicated, such as BTBs, whereas others may still be shared, such as caches. While shared caches may suffer occasionally from additional misses inflicted by context executing a kernel task, they remain warm with regard to the application since it continues to execute. Overall, the AMP design bears the potential for increased throughput that can be attributed to a reduction in both horizontal and vertical waste (*i.e.*, increased resource parallelism and cycle utilization) as well as cache benefits [23].

Exploiting the AMP mode for interrupts requires additional modifications to the operating system. Linux separates interrupt handling into a short "top half" handler and a "bottom half" routine for longer processing activity in response to an event [11]. If the interrupt occurred during an idle period, the bottom half is serviced immediately. If, however, a task was interrupted, the bottom half is enqueued as a kernel task for later processing within the "kernel idle task" (aka. `ksoftirqd`). Should the bottom half be delayed too long, its execution may be promoted to avoid buffer overflows.

We exploit the interrupt-handling mechanism in Linux specifically to cater to the AMP mode. When running in AMP mode, bottom halves of interrupts are bound to the context assigned to kernel tasks. This allows the system to execute the interrupt bottom halves concurrently with a running user task without ever interrupting the latter. The details of our modified interrupt handling mechanism are given in the following.

3. Implementation Details

We implemented our design of an AMP scheduling mode and the corresponding support for interrupt handling within bottom halves within the Linux 2.6.8.1 kernel. Prior to our modifications, this kernel already provided scheduler optimizations for a single run queue per physical processor as well as an optimized idle loop for SMT processors.

3.1. Linux Scheduler

Changes to the Linux scheduler closely resemble our design previously discussed in the context of Figure 1. Just before the scheduler transfers control to the next task to be executed, the enhancements depicted in Figure 2 are triggered.

```
if (AMP_ISON && AMP_THISCPU)
  if (next→mm != NULL) {
    cpu_clear(cpu, next→cpus_allowed);
    next = rq→idle;
    goto switch_tasks;
  }
```

Figure 2. Actual scheduler modifications

The first conditional guards the statements below to ensure that AMP mode is enabled and the current CPU is the one dedicated to kernel tasks, only. The second conditional checks if the next task selected is a user task. In this case, the CPU mask is set on the user task so that it will not be considered for later execution on this CPU. The last two lines select the idle task to be executed next on this context before jumping directly to the context switching code. This approach is customary for selecting an alternate task as the idle code immediately checks for runnable processes before idling. Hence, we ensure that the selection code (before our code shown in Figure 2) for another runnable task is properly re-run.

3.2. Redirecting SoftIRQs

As previously mentioned, the top half interrupt processing system is left unchanged. This simplifies the modifications to the system, as it limits the interaction with device drivers and other hardware-related code. Due to the dual-phase nature of the interrupt system, this is of little consequence to our performance measurements since the top halves of interrupts are designed to complete within a very short amount of time, thereby minimizing their impact on the preempted task. Also, the latency of many parts of the system may depend on the speed at which the top halves execute. We did, however, modify the behavior of the bottom-half interrupt processing system.

The primary method for handling interrupt bottom-half processing in Linux is through the use of *softirqs*, *i.e.* a unique number that identifies the bottom-half handler. *Softirqs* are registered when device drivers register their top half Interrupt Service Routine (ISR) with the operating system. Upon raising a *softirq*, its handler is assigned for processing on one of the processors in the system. While the top half remains unaltered, the logic of the *softirq* processing routine, (`__do_softirq()`), is enhanced.

On an SMT, both contexts routinely execute the `__do_softirq()` routine, which we modified as follows. When AMP mode is enabled, the context dedicated to user tasks only runs *softirqs* that match a mask set dynamically at runtime. The mask allows us to retain certain *softirqs* in the user context since selected kernel tasks, such as the timer handler, must be run on the context handling the top half to retain operational integrity of the system. Nonetheless, the majority of the *softirqs* can be diverted to the kernel context by masking them in the user context. Hence, the kernel context processes all pending *softirqs* and, in addition, collects *softirqs* that were raised on the user context. To safely divert *softirqs*, locking was required for raising and clearing *softirqs*.

Linux also provides another mechanism for sequential queuing and execution of lower-priority bottom halves, so-called *tasklets*. The tasklet mechanism simply chains scheduled work together and processes it se-

quentially instead of observing the priority-based handling of *softirqs*. In fact, tasklet processing is invoked from the lowest-priority *softirq*. Similar modifications were made to the tasklet system to ensure that all were processed by the dedicated kernel context.

3.3. AMP Control Module

Support for the AMP scheduling mode was provided by a control module that can be dynamically loaded. Its primary purpose is to facilitate the transitioning from non-AMP mode to AMP mode. In addition, it adjusts the *softirq* mask for the user context. Insertion of the module immediately transitions to AMP mode, and an interface to the */proc* filesystem is provided for adjusting the *softirq* mask for the user context. The module also provides diagnostic information, which we use to observe the functionality and performance of our modifications to the interrupt system. Loading the module into the kernel at runtime effectively enables the switching between non-AMP mode and AMP mode from user space.

4. Experimental Framework

We designed a number of experiments to assess the merits of our implementation of AMP scheduling on an Intel Pentium 4 with SMT (Hyperthreading) support. In the experiments, the AMP mode is compared to conventional scheduling on the same hardware with two flavors for a total of three scheduling variants.

1. Uniprocessor (UP) mode treats the SMT processor as a normal uniprocessor, *i.e.*, the second hardware context of the SMT is disabled to make all resources available to one context, only.
2. Simultaneous multiprocessing (SMP) mode is the default scheduling policy to handle multiple processors under Linux. In this mode, a multi-context SMT processor is treated as a conventional shared-memory multiprocessor, *i.e.*, subtleties of SMTs are mostly ignored. It shall be remarked that the Linux SMP support incorporates some specializations to reflect the differences specific to an SMT, as mentioned in Section 3, but these are minor compared to our AMP design.
3. AMP provides the novel abstraction of a dedicated context for kernel tasks besides another context for user tasks previously introduced in this paper.

We evaluated the impact of these schedulers on performance for a subset of the SPEC 2000 benchmarks [2, 1]. The benchmarks chosen were a mix of floating-point and integer programs that would fit completely into a quarter of our available memory without the need for paging.

First, we assess the completion time for n tasks (threads) of a benchmark (identical executions of the same application), where n varies between two and four. The objective of this experiment is to assess if the parallelism of SMTs can be exploited in SMP mode when tasks dispatch instructions in parallel as opposed to serializing their execution in UP or AMP mode. The second context in AMP mode remains un-utilized in these experiments. The special case of just one thread ($n = 1$) was excluded since it would not trigger any parallelism at the SMT and, hence, does not meet our objective.

Second, we repeat the same experiment, but we also trigger kernel activity by imposing considerable network traffic in fast succession on the system. This illustrates how well kernel activity can be sustained in the different modes. In UP and SMP modes, user tasks are interrupted and upper as well as lower halves of interrupts are executed. With AMP, virtually all lower halves are handled in the kernel context while the execution of the benchmark proceeds in the other context without interruption.

Third, we assess benefits of dynamically switching modes depending on the "best fit" of a mode for a given application in the presence of network (kernel) activity. This illustrates the potential of our AMP mode for integration with the SMP mode. The scheduler could then be enhanced to execute applications suitable for one mode while queuing others that excel in another mode. At short intervals, modes and queues could then be swapped, potentially preempting any running tasks and, thus, deferring them to the next mode change.

The network activity imposed in these experiments follows a client-server messaging paradigm where the server resides on the benchmarked platform while the client is located on a separate computer. The protocol resembles common client-server interactions where large blocks of data are continually being requested by client(s) and supplied by the server. This resembles the I/O-bound load that simple web servers or customized data-feedback applications generate. Client requests are relatively small while server responses are much larger in size.

The experiments are specifically designed to ensure that the desired number of threads (benchmark tasks) were running at all times. Consider the case of four threads launched simultaneously to execute an application. A small control wrapper around the application code ensures that there are always four threads present when assessing the completion time over all

four threads. This is depicted in Figure 3. The wrapper spawns a fixed number of threads of the same application. When a thread completes, another thread is started immediately so as to guarantee a constant load, *i.e.*, threads run with identical resource contention during our measurements when exploiting the SMT architecture.

```
fork n times;
if (parent) {
  while (number_complete ¡ n) {
    wait_for_child_exit();
    re-spawn a new child;
  }
} else {
  sched_yield();
  execute benchmark;
}
```

Figure 3. Multi-Thread execution algorithm

The `sched_yield()` ensures that new tasks forfeit their right to execute immediately, which gives other threads a chance to be spawned before execution proceeds. The timing measurements only reflect the completion times of the first n threads, *i.e.*, four threads in our the example. More specifically, the overall completion time of n threads spans the period from creating the threads to the completion of the last (slowest) of the threads. The runtime overhead of re-spawned threads is not included. This re-spawning technique is crucial for consistent results and has been used before in SMT experiments [23]. If we did not re-spawn, three of the four threads may finish early while the remainder of execution for the last thread would proceed at a faster pace due to reduced resource contention. Average iteration times per thread were also measured and are discussed in a final set of experiments to assess the predictability under the different modes. The overall completion time provides a fair performance metric since we want to compare work under a constant load scenario.

5. Experimental Results

We conducted a number of experiments to assess the benefits of asymmetric multiprocessing, as outlined in the previous section. First, each benchmark is considered in isolation for (a) UP, (b) SMP and (c) AMP modes in (1) absence and (2) presence of kernel activity. Second, execution of different benchmarks is considered under dynamic mode switching between AMP and SMP. Third, the performance predictability is assessed for the different modes.

5.1. Homogeneous Workload Tests

Tables 1 and 2 depict the performance results for the SPEC benchmarks ART and MESA, respectively. These benchmarks are representative for the behavior of the other SPEC benchmarks tested. Hence, we will first discuss these results in detail before presenting the results for all tested benchmarks. For each benchmark, we tested the completion time for the three modes UP, SMP, AMP. Within each mode, we benchmarked configurations with two, three and four threads (2T, 3T, 4T). We depict the completion time (CT) in seconds (Columns 3 and 5) and their change to the corresponding UP configuration. We also distinguish the absence of kernel tasks or network activity (Columns 3 and 4) and its presence (Columns 5 and 6).

		Without Net		With net	
Mode	Config	CT	%-Change	CT	%-Change
UP	2T	84.67	0.00%	95.67	0.00%
	3T	129.33	0.00%	146.33	0.00%
	4T	170.00	0.00%	192.00	0.00%
SMP	2T	86.00	1.55%	103.33	7.41%
	3T	159.00	18.66%	179.00	18.25%
	4T	173.33	1.92%	207.67	7.55%
AMP	2T	85.00	0.39%	88.00	-8.72%
	3T	125.33	-3.19%	135.33	-8.13%
	4T	170.33	0.19%	176.33	-8.89%

Table 1. Art Runtime [sec] and Change to UP

		Without Net		With Net	
Mode	Config	CT	%-Change	CT	%-Change
UP	2T	327.67	0.00%	378.00	0.00%
	3T	491.67	0.00%	576.67	0.00%
	4T	659.00	0.00%	777.33	0.00%
SMP	2T	438.67	25.30%	286.67	-24.16%
	3T	426.33	-15.33%	492.67	-14.57%
	4T	520.33	-26.65%	574.67	-26.07%
AMP	2T	340.67	3.82%	367.67	-2.81%
	3T	518.33	5.14%	549.67	-4.91%
	4T	702.33	6.17%	743.67	-4.52%

Table 2. Mesa Runtime [sec] and Change to UP

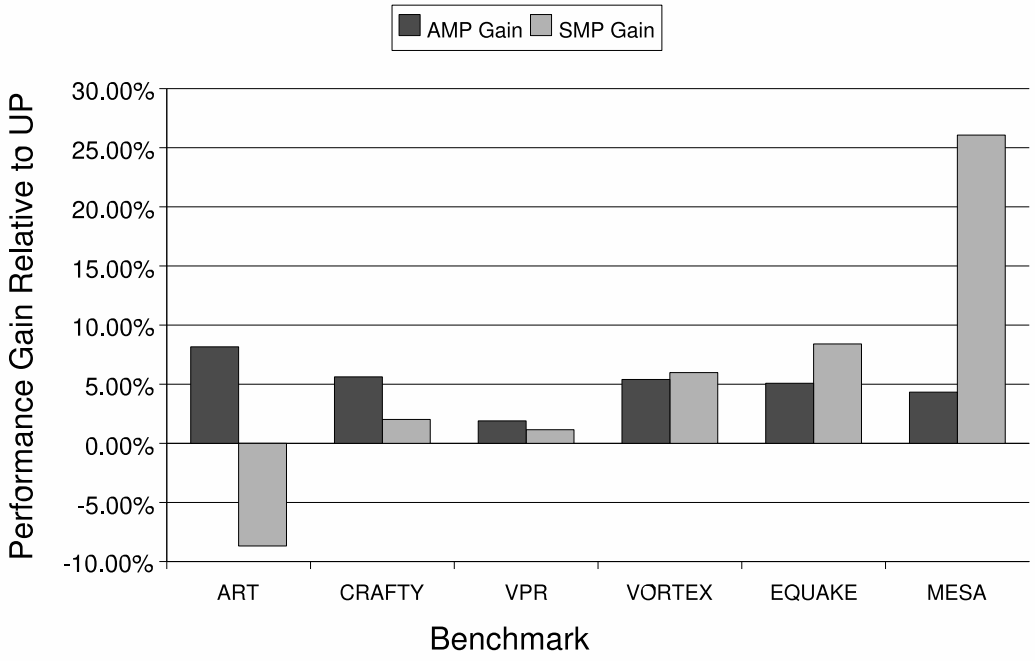

Figure 4. Improvement of SMP and AMP modes over UP for 4 Threads (with Network Activity)

The results for ART in Table 1 illustrate that, in the absence of network traffic, UP outperforms SMP marginally while AMP remains mostly unaffected (small win or loss). However, when network traffic is added, SMP suffers performance losses of 7-18% while AMP gains 8% in speed. Notice that an odd number of threads (3T) gives significantly different results than an even number of threads (2T and 4T). This anomaly will be discussed later. Overall, ART favors a uniprocessor mode (UP or AMP) over a multiprocessor mode. ART is a floating-point intensive benchmark that efficiently utilizes the resources with a single benchmark thread. Additional threads result in an overall reduction of performance due to resource contention.

In contrast, the results for MESA in Table 2 illustrate that, in the absence of network traffic, SMP sometimes outperforms UP while AMP lags behind UP. When network traffic is added, SMP always outperforms UP by 14-26% while AMP gains only up to 5% in speed. Again, results for odd numbers of threads are explained later. However, we cannot explain the 25% decrease in performance for two threads in SMP mode, which is reproducible. Overall, MESA favors a multiprocessor mode over a uniprocessor mode. MESA is an integer-centric benchmark that does not fully utilize resources from a single thread, *i.e.*, a second thread of the same workload still results in better resource utilization.

It is not imperative to understand the cause of better or worse performance for any specific benchmark in our work. Instead, we need to be able to observe if a benchmark favors uni- or multiprocessors to decide upon scheduling modes, as will be shown later. Of the benchmarks tested, ART and MESA represent our best performers in the presence of network traffic for AMP and SMP, respectively. We found that it is sufficient to compare the behavior of benchmarks between AMP and SMP mode in the presence of network activity irrespective of the number of threads.

Figure 4 depicts the results for all tested benchmarks under network traffic for four threads. Improvements in completion time are reported for AMP and SMP relative to UP. The first three applications see a larger benefit when running in AMP mode compared to SMP mode. The last three applications, however, benefit more from SMP mode. Notice that the workload was not decisive for the behavior of a benchmark. Some floating-point benchmarks favor AMP, others excel under SMP (and similarly for integer benchmarks).

The detailed results from the tested SPEC benchmarks for different number of threads and presence/absence of network traffic lead us to the following conclusions. First, we observe that AMP mode does, in fact, behave almost identically to UP mode in the ab-

sence of network activity. Second, AMP mode always outperforms UP mode in the presence of network activity. Hence, we can classify the benchmarks into two categories: one that favors multiprocessors, called pro-SMP, and one that favors uniprocessors, referred to as pro-AMP. The naming of the latter is justified because AMP always performs as well or better than UP while exhibiting many of the same characteristics. The assignment to categories is shown in Table 3. Interestingly, the pro-AMP category is also the pro-UP

Category	Member Benchmarks
pro-AMP	Vpr, Crafty, Art
pro-SMP	Vortex, Mesa, Equake

Table 3. Categorization of Benchmarks

category when ignoring network activity. Thus, applications that perform better on a uniprocessor than in SMP mode also perform better in AMP mode, even when network activity is present. Thus, adding the AMP functionality to the operating system provides the ability to utilize the additional resources and capabilities provided by the SMT processor without running in a normal SMP mode when such a mode is detrimental to performance of a specific application.

5.2. Hybrid Tests

Our modifications to the operating system extend beyond a single, static choice of scheduling mode to where dynamic transitioning between AMP and SMP mode can be accomplished. Since neither AMP nor SMP modes offer the best performance by themselves for different benchmarks, the scheduler can then choose the most appropriate mode for an application. We designed a hybrid SMP+AMP mode that executes one homogeneous benchmark in one mode, issues a mode change and then runs the other mode. The performance of the hybrid mode is then compared to the completion time over both benchmarks for any single mode (UP, SMP and AMP).

Our AMP-modified kernel was originally derived from an SMP kernel. Hence, disabling AMP mode causes it to behave exactly like the SMP mode. Thus, we can observe the performance of a hybrid run by simply combining the appropriate results from previous runs. We conducted experiments to run a pro-SMP benchmark, inserted the AMP control module described in the implementation and then executed a pro-AMP benchmark. After excluding the (small) mode switch overhead, the overall completion time matches the sum of the individual benchmark times for the respective modes. Hence, one can reliably derive the hybrid data constructively from data of benchmarks in their corresponding single mode, which would facilitate an automated choice of modes by the scheduler. The hybrid mode allows one to select the fastest run of each application, which indicates the best mode for each application. The result is then known to be in the smallest possible overall sum.

Consider the results for sequential executions of Crafty and Equake depicted in Table 4. The overall runtime in hybrid mode, a combination of Crafty in AMP mode and Equake in SMP mode, is the lowest possible value (including any single-mode as well as any other hybrid run).

Mode	Crafty	Equake	Total
UP	462.67	441.33	904.00
SMP	453.33	**404.33**	857.66
AMP	**436.67**	419.00	855.67
Hybrid	436.67	404.33	**841.00**

Table 4. Hybrid Times [sec] for Equake and Crafty (4 Threads plus Network Activity

Another example is depicted in Figure 5 for Art and Mesa. Art excels under AMP while Mesa peaks in SMP mode with a 22.2% gain over UP, which outperforms the 18.9% gain of the second-best choice (SMP-only).

Mode	Art	Mesa	Total
UP	192.00	777.33	969.33
SMP	207.67	**574.67**	782.34
AMP	**176.33**	743.67	920.00
Hybrid	176.33	574.67	**751.00**

Table 5. Hybrid Times [sec] for Art and Mesa (4 Threads plus Network Activity)

We then determined the overall completion time for various combinations of two benchmarks where one benefits from AMP while the other excels in SMP mode. We planned to test this simple hybrid strategy by choosing two applications, one of which benefited more from SMP mode while the another benefits more from AMP mode. A comparison of all projected hybrid runtimes is given in Figure 5. The overall runtime overhead is shown for AMP and SMP modes as well as the hybrid combination. The chart illustrates clearly that the hybrid solution

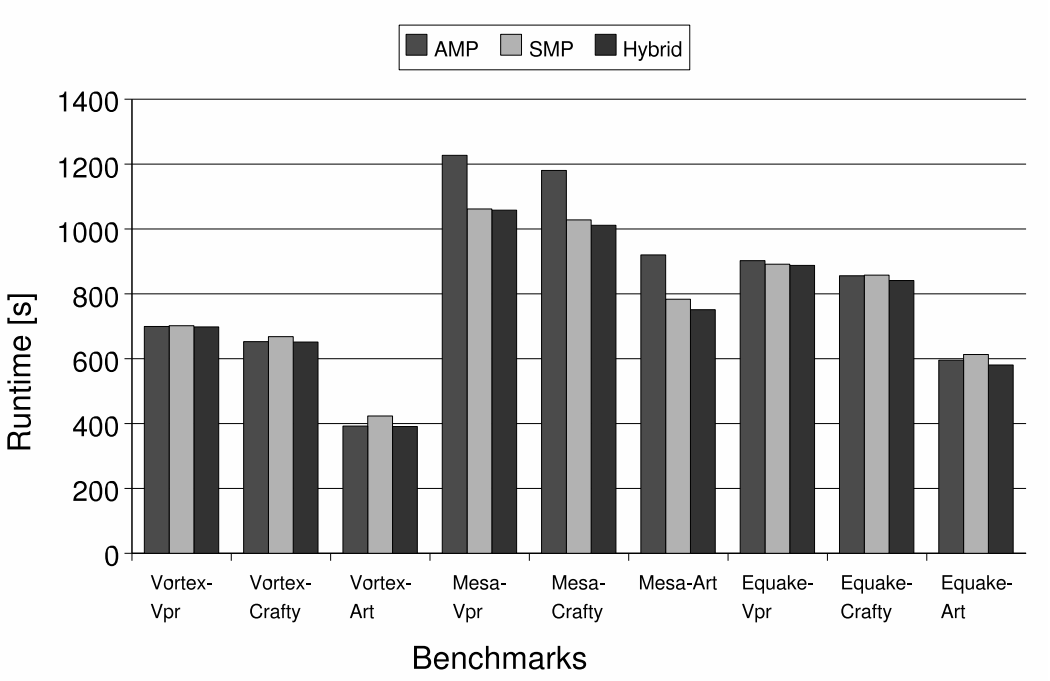

Figure 5. Hybrid Runtimes [sec]

always outperforms any single choice of mode when applications favor opposite modes. In some cases, however, the difference between a hybrid and static modes is so small that the benefit may not justify a mode switch. In others, a significant improvements are seen. In practice, the challenge for the hybrid solution is to balance the dynamic workload between both modes without affecting the responsiveness of tasks when hybrid is implemented with separate run queues, as suggested in our implementation.

5.3. Performance Predictability

We next conducted experiments to assess the variance in execution time under different modes of execution, both with and without network traffic. Table 6 shows results for a dual-threaded microbenchmark with a workload in the inner loop that is sampled 9969 times in an outer loop. Specifically, the workload consists of multiple independent floating-point (FP) calculations, which should provide sufficient instruction-level parallelism to utilize all FP resources in a single thread. The columns of the table depict the average runtime (in seconds), standard deviation, minimum and maximum for the inner loop of the benchmark followed by the average runtime for the entire benchmark. These numbers are reported with and without network activity (Columns 4-8 and 9-13, respectively) as well as AMP and SMP modes. For each mode, three runs of two threads each are measured.

The results in the table show that runtimes are very predictable in the absence of network traffic since the standard deviation is constantly low for both AMP and SMP. When adding kernel activity (network traffic), the standard deviation for AMP only increases insignificantly. Hence, AMP remains predictable with network traffic. In contrast, the standard deviation for SMP increases by an order of a magnitude. Thus, SMP becomes significantly less predictable in the presence of network activity.

These findings illustrate the benefits of AMP in increased responsiveness and predictability of application performance in the presence of high interrupt arrival rates. Hence, our synergistic OS/hardware approach is not only beneficial for alternating executions of applications with different SMT characteristics, it also provides sustained responsiveness and predictability during unexpected and frequent interrupts.

5.4. Anomaly for Odd Number of Threads

We already indicated in the context of discussing Tables 1 and 2 the existence of any anomaly for odd number of threads. Specifically, large discrepancies were observed for single completion times of any of the three threads while the average time over the threads remained relatively constant and close to our expectations. This

			Without Net					With Net				
Mode	Trial	Thread	Runtime	St.Dev.	Min	Max	Average	Runtime	St.Dev.	Min	Max	Average
	1	1	76.34	13.85	7.46%	0.57%	3822.68	81.01	50.19	9.42%	5.79%	4045.03
	1	2	76.57	16.58	7.32%	0.49%	3823.90	81.08	49.59	6.90%	4.62%	4044.85
	2	1	76.45	15.55	7.12%	0.52%	3822.79	81.00	48.63	6.86%	5.15%	4047.25
AMP	2	2	76.56	14.47	8.57%	0.56%	3822.59	81.04	48.48	6.95%	4.81%	4047.56
	2	1	76.12	15.87	7.26%	0.56%	3822.54	80.96	48.55	6.67%	5.95%	4049.84
	3	2	76.55	14.66	7.05%	0.56%	3822.56	81.19	48.26	7.22%	4.46%	4049.79
	Avg		76.43	15.16	7.46%	0.54%	3822.84	81.05	48.95	7.34%	5.13%	4047.39
	1	1	85.62	38.02	2.26%	4.52%	8552.55	84.67	545.15	15.13%	6.23%	8452.41
	1	2	85.61	38.42	5.52%	2.56%	8556.65	87.23	584.67	10.32%	12.32%	9034.39
	2	1	85.60	38.50	5.25%	4.60%	8559.44	83.93	486.67	16.11%	5.69%	8377.29
SMP	2	2	85.60	37.84	5.31%	3.70%	8550.38	87.22	532.10	8.20%	13.14%	9123.99
	3	1	85.61	37.28	2.29%	4.69%	8551.96	86.99	120.89	4.86%	9.71%	9406.63
	3	2	86.61	38.32	5.39%	3.15%	8555.79	81.63	68.36	5.68%	3.06%	8150.30
	Avg		85.78	38.06	4.34%	3.87%	8554.46	85.28	389.64	10.05%	8.36%	8757.50

Table 6. Runtimes of Microbenchmark [sec]

caused the completion time to vary as it encompasses the period from start of all threads to completion of the last thread. This overall completion time was always significantly greater than the average. Since an even number of threads does not exhibit this behavior, we hypothesized that the cause was rooted in scheduling details for an odd number of threads. Experiments with five threads reaffirmed our suspicion as they yielded the same anomalies and lead us to believe that measured completion times are not reliable for an odd number of threads.

Upon further examination, it was discovered that the standard Linux scheduler is not completely fair to each of the three threads. The threads appear to be sequentially pinned to CPUs. Hence, an even number of threads (or, more accurately, a thread count which is a multiple of the number of CPUs in the system) results in an evenly distributed load. Conversely, an odd number of threads inflicts an uneven load. In such a case, all processes should be scheduled round-robin on *any* available SMT contexts to ensure a fair time distribution. There simply is no benefit to dispatch a preempted context to its original context in an SMT since even L1 caches are shared and other resources, such as BTBs, are likely invalidated due to context switching. We believe that the current Linux scheduler requires round-robin enhancements without CPU pinning within contexts of an SMT.

6. Related Work

Applications rarely fully utilize modern processors. This is often due to the latency of other devices within a system, upon which the executing application depends. For in-order pipelined architectures, multithreading provides increased processor utilization by multiplexing the available resources among multiple instruction streams. The result is not a faster execution time for a single application, but rather an increased total throughput for multiple applications. Early multithreaded architectures, such as the MIT Alewife machine [3] and the Tera computer [7] provided fast hardware context switching, which allowed the pipeline to be loaded with an instruction from a different stream every cycle without penalty. Interactions between the running threads were minimal because only one instruction occupied a given pipeline stage at a time.

The introduction of superscalar architectures brought concurrent execution of multiple instructions from a single stream. Sufficient instruction-level parallelism (ILP) identified at runtime by the CPU increases throughput by processing multiple instructions per cycle in a single stage. Such an architecture is limited by the finite amount of ILP available in any instruction stream. Tullsen *et al.* describe this problem in terms vertical and horizontal waste [24].

Vertical waste occurs when the CPU issues no instructions for a given cycle. Horizontal waste occurs when the CPU is unable to fill all available issue slots in a cycle, which is a result of low thread ILP. Simultaneous Multithreading focuses specifically on reducing horizontal waste by giving the CPU additional instruction streams from which it can fill the remaining issue slots. This can, however, simultaneously combat vertical and horizontal waste: a given cycle may be filled with instructions from any one of the current threads, or it may be filled with a combination of instructions from more than one thread.

6.1. Symbiotic Job scheduling

A simultaneously multithreaded processor introduces many complexities and unique characteristics over a similarly-equipped single-threaded processor. Most conventional multiprocessor systems replicate entire CPUs to achieve parallelism. Such a system requires an operating system to intelligently schedule processes to optimize cache performance and reduce false sharing. Typically, an operating system would ignore things such as the functional unit usage profile of concurrently executing applications, as the independent processors do not share these resources. These factors are exactly reversed in the case of an SMT processor and provide the justification for specific operating system support for SMT CPUs. When co-scheduling two tasks on the virtual CPUs of an SMT, false sharing can actually increase performance (since the contexts share a cache), and processes need not be rescheduled on the same CPU, as the shared cache is warm no matter which context executes the process. Similarly, functional units are shared, and, thus, the operating system could increase performance by intelligently scheduling processes that would minimize interference.

Snavely *et al.* use the term *symbiosis* to describe the effectiveness with which multiple jobs achieve speedup when co-scheduled on a simultaneously multithreaded processor [19]. They describe a system, called SOS, that dynamically determines the most beneficial schedule for a set of tasks, based on their measured performances. This is referred to as *symbiotic job scheduling* [21] and is a crucial step in improving performance on an SMT system. They take their idea further [20] by introducing weights into their system, providing the (very important) ability to support QoS within the system. This is a key concept, as it is similar to our work, although it assumes that the operating system is always scheduled with the highest priority. Also, the SOS system spends execution time measuring the performance of many combinations of running tasks before determining the best schedule. Another solution, however, could be to classify applications statically, thereby eliminating the need for runtime sampling and preventing short-lived processes from hurting the performance of the system.

Settle *et al.* propose hardware support through active performance counters that can then steer scheduling decisions in the operating system based on different memory behavior of threads [18]. Co-scheduling of threads with non-competing memory behavior is shown to enhance the overall throughput. Our work differs from any of the above approaches by promoting an asymmetric scheduling solution and does not require any hardware enhancements to existing processor designs.

6.2. Other Asymmetric Systems

The idea of a single system with multiple processors dedicated to separate duties has been in practice since at least 1964. The Control Data Corporation (CDC) 6000 series computers used this technique to allow I/O parallelism [8]. One or two central processors were used to perform high-speed arithmetic operations and general program execution, much like the CPUs of today's microcomputers. Additionally, small peripheral processors (PP) provided I/O service between external devices and central memory (CM). The PPs each had small private memories and access to all locations of the CM. By loading a small program into a PP's private memory, a complex I/O operation could be carried out while allowing the central processor to continue execution of other tasks. Thus, the CDC 6000 machine had multiple processors dedicated to different tasks executing different code. Much of the CDC operating system was implemented in PP code,[1] resulting in a system where most of the OS was executed on dedicated processors, minimizing interference with user tasks.

Nahum *et al.* studies the effect of adding packet-level parallelism by exploiting shared-memory multiprocessors, specifically in terms of protocol overhead, such as locking, for TCP and UDP enhancements [14]. Their objective was to obtain packet-level parallelism, which was accomplished but limited in terms of the amount of parallelism realized. Muir and Smith designed an asymmetric multiprocessing extension called Piglet, which was integrated with Linux [13]. Piglet provides a frameset abstraction to multiplex device requests from multiple processes to a single device observing QoS restrictions, and this multiplexing is realized at the driver level so that it can be decoupled from the application. In their experiments, network requests could such be served on a different processor than the application within a two-way SMP. Rangarajan *et al.* designed a system called TCP Server that offloads TCP traffic onto dedicated SMP processors or even different processing nodes within a cluster [16]. Their objective was to off-load the even-increasing processing demands due to high-bandwidth network connections. Interrupt was replaced with polling, and buffering could be avoided in some instances due to the dedication of a processor to network traffic. Interrupts were masked on the application side by reprogramming the hardware at the APIC level. These approaches differ from our

1 The PPs used a different ISA from the central processor and, therefore, required code to be specially written.

work in that we do not divert interrupts at the hardware level, provide packed-level parallelism or off-load network traffic to another physical processor. Instead, we divert *Softirqs* to a dedicated logical context on the same physical processor with shared resources. We focus on performance benefits of multi-processing in different processor modes, and our hybrid mode is unparalleled to past work, particularly since it bears the potential to use multiple context whereas past work kept one processor idle when no network traffic required processing. Our efforts are also aimed at providing a tighter coupling between the application and interrupt processing than possible in SMPs, which includes but is not limited to network traffic. Hence, kernel tasks in one context can place packets in L1 cache so that applications can benefit from data locality due to network traffic.

A more modern example of an asymmetric architecture is the IBM BlueGene/L machine (BGL) [22]. The BGL system packages two PowerPC 440 CPUs per processing element (PE). Unlike the CDC 6000 system (in which the CPUs were not similarly-equipped), a BGL PE has two equally-powerful processors in a pseudo-SMP configuration.[2] Although the system is capable of symmetric task assignment to both processors on a PE (the so-called virtual mode), normal operation utilizes one CPU for computation and the other for messaging (the so-called co-processor mode). The BGL machine is an example of an SMP system where tasks are not symmetrically assigned. Effectively, the second physical processor becomes a service or co-processor, similar to the intent of our work with respect to an additional context of an SMT processor. However, many large-scale benchmark perform better in virtual mode, which defaults to the symmetric model, even though BGL was originally not designed for this type of use.

7. Conclusion

In this paper, we explore an alternative way to approach utilization of the additional resources provided by an SMT processor. We depart from traditional symmetric scheduling techniques and suggested an operating system modification that illuminates a new path in light of the diminishing returns that result from conventional symmetric multiprocessing. We achieve this goal while maintaining compatibility with existing algorithms as well as the added benefit of being able to utilize both existing and new paradigms in the same system at runtime.

Our research verified that some applications perform better when scheduled alone, as opposed to being co-scheduled on an SMT processor. We tested benchmarks and categorized their behavior into those which do and do not perform well on an SMT. We also showed that our modification to a conventional operating system allows poor SMT performers to realize a performance gain. Finally, we showed that the flexibility of our system can achieve a best-of-both-worlds result when faced with a heterogeneous application workload.

References

[1] Cfp2000 benchmark descriptions, 2000.

[2] Cint2000 benchmark descriptions, 2000.

[3] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT alewife machine: Architecture and performance. In *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture (ISCA'95)*, pages 2–13, 1995.

[4] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, 1988.

[5] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the International Symposium on Computer Architecture*, pages 104–114, June 1990.

[6] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the1990 International Conference on Supercomputing*, pages 1–6, 1990.

[7] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the1990 International Conference on Supercomputing*, pages 1–6, 1990.

[8] Control Data Corporation. *6400/6500/6600 Computer Systems Reference Manual*, h edition, 1969.

[9] N. Gloy, C. Young, J. B. Chen, and M. D. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *Proc. 23rd Annual Intl. Symp. on Computer Architecture*, pages 12–21, 1996.

[10] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85, September 1997.

[11] R. Love. *Linux Kernel Development*. Sam's Publishing, 2004.

[12] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):1–12, February 2002.

[13] S. Muir and J. Smith. Functional divisions in the piglet multiprocessor operating system. In *8th European SIGOPS Workshop*, pages 255–260, 1998.

2 The PowerPC 440 lacks actual SMP hardware, so the processors on a PE are not L1 cache-coherent.

[14] E. M. Nahum, D. J. Yates, J. E. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation (OSDI'94)*, pages 125–137, 1994.

[15] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, 1996.

[16] M. Rangarajan, A. Bohra, K. Banerjee, E. Carrera, R. Bianchini, and L. Iftode. Tcp servers: Offloading tcp processing in internet servers. design, implementation, and performance. DCS-TR 481, Rutgers University, Mar. 2002.

[17] J. Redstone, S. J. Eggers, and H. M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Architectural Support for Programming Languages and Operating Systems*, pages 245–256, 2000.

[18] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural support for enhanced smt job scheduling. In *International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[19] A. Snavely. Explorations in symbiosis on two multithreaded architectures, 1999.

[20] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor, 2002.

[21] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[22] T. B. Team. An overview of the bluegene/l supercomputer. Technical report, IBM and Lawerence Livermore National Laboratory, 2002.

[23] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages ??–??, 1995.

[24] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 1995.