CPU Resource Reservation for Simultaneous Multi-Thread Systems

Hiroshi Inoue, Takao Moriyama, Yasushi Negishi, and Moriyoshi Ohara
IBM Tokyo Research Laboratory
{inouehrs, moriyama, negishi, ohara}@jp.ibm.com

Abstract

Simultaneous Multi-Thread (SMT) techniques are becoming popular because they increase the efficiency of CPU resource usage by allowing multiple threads to run on a single physical processor at a very fine granularity. Emerging real-time applications, however, may not benefit from the SMT techniques because those techniques often compromise the predictable performance characteristics of applications, which real-time applications typically need to meet their computation deadlines. In this paper, we propose a new resource reservation scheme for SMT systems. In this scheme, a task scheduler dynamically enables and disables the SMT facility while real-time applications are running by monitoring the progress of the real-time applications. In this way, real-time applications can still meet their computation deadlines, and other best-effort applications can gain a high throughput due to the SMT facility. We have implemented this scheme on a Linux kernel and evaluated it on a Hyper-Threading processor, an Intel's implementation of SMT techniques. Our experimental results have shown that, for our workload, our scheme can guarantee real-time applications to use reserved resources while best-effort applications can obtain a high throughput due to the SMT facility.

1. Introduction

Soft real-time applications, such as multimedia applications, are becoming increasingly important for general-purpose systems. These applications often need to use certain amount of computation resources in order to complete their computations by a certain deadline. Thus, Operating systems need to allocate resources to meet various resource requirements for each process in the system. In fact, previous works [1-5] have addressed task scheduling schemes that meet such resource requirements by allowing each real-time process to declare its worst-case computation time and the deadline.

Those scheduling schemes, however, may fail to allocate resources as applications require on a processor with an SMT (Simultaneous Multi-Thread) [6] facility. This is because those scheduling schemes assume the worst-case calculation time is predictable and because this assumption is no longer true for SMT systems. Since the SMT facility allows multiple threads to share one processor at a very fine granularity, such as a processor-cycle level, those threads compete each other for various resources such as execution units, L1/L2 caches, bus, and register file. The performance of each thread, therefore, heavily depends on the behavior of other threads on the same processor. In reality, such inter-dependency has already existed even for non-SMT processors due to conflicts of shared resources, such as cache lines. The degree of the inter-dependency, however, is much higher in SMT systems than in non-SMT systems. In other words, the CPU time of a process less accurately indicates the amount of consumed computation resources (i.e. the number of instructions executed) on SMT systems than on non-SMT systems. Therefore, existing CPU resource reservation systems that reserve CPU time for real-time processes may not work as intended on SMT processors.

One can disable the SMT facility to avoid the effect of other threads on the same processor while a real-time process is dispatched. While this naive method can meet the time constraints of real-time processes, however, this method reduces the overall throughput that we could obtain by using the SMT facility. Because many general-purpose systems need to run real-time and best-effort (non-real-time) processes at the same time, it would be ideal if we could achieve two goals - high throughput for best-effort processes and guaranteed resource allocation for real-time process - by exploiting the SMT facility.

To achieve these two goals, we propose a new resource reservation scheme for SMT systems. In this scheme, the task scheduler keeps track of the progress of each real-time process and dynamically enables/disables the SMT facility. More specifically, our method allows best-effort processes to run on an SMT processor as

long as they do not prevent the real-time processes from meeting their deadlines. We implemented and evaluated this scheme in a Linux kernel on a real SMT processor. Our experimental results have shown that our scheme can achieve the above-mentioned two goals for our workload; our scheme can guarantee real-time processes to meet their deadlines, and our scheme can maintain higher throughput than a naive scheme that disables the SMT facility while a real-time process is running.

The rest of the paper is organized as follows. Section 2 discusses related works. Section 3 describes our resource reservation scheme, called *slack time monitoring*. Section 4 describes our implementation of this scheme on a Linux kernel. Section 5 discusses our experimental environment and experimental results. Section 6 discusses future works. Finally, Section 7 summarizes our work.

2 Related works

CPU resource reservation is a technique to ensure a minimum execution rate. One form of the reservation is a specification such as *"reserve 10 ms of time out of every 50 ms for the MPEG encoding process"*. Resource Kernel [1, 2] is one of the most widely known resource reservation systems, for various kinds of resources, such as processor and network bandwidth. Resource Kernel guarantees allocations of CPU time to real-time processes according to declared periods of processing and required CPU time within a time period. The existing implementation of Resource Kernel works well on non-SMT processors, though it may not work on SMT processors as intended. This is because the progress of a thread heavily depends on that of other threads on the same SMT processor, and programmers cannot predict the behavior of the other threads. This issue arises not only for existing reservation-based real-time scheduling schemes [1 - 4], but also for other real-time scheduling schemes such as rate monotonic scheduling [5].

SMT-aware task-scheduling schemes (such as [7 - 12]) had been proposed. Snavely *et al.* [7, 8] proposed SOS, which selects the co-scheduled processes (i.e. running at the same time on the same SMT processor) in such a way to increase the overall throughput. They reported that SOS has improved not only the throughput but also the response time. Parekh *et al.* [9] proposed similar approach to enhance throughput by selecting the appropriate set of processes to co-schedule. Nakajima *et al.* [10] proposed a scheduling scheme for an SMP of SMT processors, which balances the load among processors. These scheduling schemes consider resource conflicts among threads on an SMT processor like ours, however, these did not address on real-time processes with time constraints. Our scheme, on the other hand, targets a set of real-time processes and best-effort processes.

Jain *et al.* [11] proposed a soft real-time scheduling scheme that reduces the side effect of SMT. This scheme is also based on an idea that selects co-scheduled processes to minimize the effect of SMT. This approach is effective in statistically reducing the number of deadline misses. This scheme, however, does not necessarily guarantee the allocation of required computation resources for real-time processes. Our proposed scheme, on the other hand, intends to guarantee real-time processes to use required resources. Our method, moreover, is complementary to the Jain's scheme, that is, an appropriate decision of job selection can enhance overall throughput in addition to the results of our method. Cazorla *et al.* [12] addressed removing an unpredictability of a performance of a thread on an SMT processor. They proposed a method to enforce giving a certain IPC (instructions per clock cycle) to a high priority thread by using the combination of the task scheduler and the special resource sharing policy on the processor. In contrast, our approach uses only a task scheduler and it can be running on the existing implementations of SMT, such as Intel's Hyper-Threading [13]. To our best knowledge, this paper is the first report that quantitatively evaluated reservation-based real-time scheduling schemes on a commercially-available SMT environment.

3. Our resource reservation scheme for SMT processors

3.1 Basic algorithm

To illustrate our scheme we present the following example, a real-time process reserves an amount of computation resource $c_{reserve}$ before its deadline at $t_{deadline}$. We represent the amount of computation resource $c_{reserve}$ by the CPU time without considering any other processes on the same processor. We call this condition the *single-threaded* condition. When a real-time process is dispatched to one thread of an SMT processor, the task scheduler first calculates the *slack time* t_{slack}. Slack time means the time that other processes can use without jeopardizing the timing

constraints of the real-time process, and can be represented as $t_{slack} = T - C$, where T is remaining time to the deadline, i.e., $T = t_{deadline} - t_0$ and t_0 denotes the current time, and C is remaining reserved amount of computation resource that the real-time process must obtain before its deadline, i.e., $C = c_{reserve} - c_0$ where c_0 denotes the amount of computation resource the process has obtained. C and c_0 are also represented by single-threaded CPU time.

If the slack time is greater than zero, the task scheduler can dispatch other best-effort processes in order to increase overall throughput. We call such processes running together with real-time processes at the same time on the SMT processor *co-scheduled processes.* While co-scheduled processes are running on the processor, the slack time must be continuously monitored to insure it is greater than zero in order to guarantee the allocation of the reserved amount of computation resource to the real-time process. When the slack time becomes zero, the task scheduler preempts co-scheduled processes, and the hardware threads of the processor except for that used by the real-time processes must become idle so as not to interfere with the real-time processes until they obtain their reserved amounts of computation.

3.2 Slack time monitoring method

Our scheme has two implementation issues: when to measure the slack time and how to do it. First we discuss the timing of slack time measurement. One of the possible ways to do monitoring of slack time is by using an external timer interrupt and monitoring the slack time in the interrupt handler of the timer. Both a fixed-interval polling timer and a variable interval timer can handle the monitoring, but the fixed-interval timer tends to suffer from high interrupt overhead, though it offers a simpler algorithm and implementation.

When a variable interval timer is used to monitor the slack time, the interval between one interrupt at time t_0 and the next one at time t_{next} can be represented by the following equation using the monitored slack time itself,

$$t_{next} - t_0 = t_{slack}\,, \tag{1}$$

where t_{slack} is the slack time measured at time t_0. Thus the next timer interrupt is set at time

$$t_{next} = t_0 + t_{slack}, \tag{2}$$

in the interrupt handler for the timer interrupt at time t_0. The flow chart of an interrupt handler for this timer interrupt is shown in Figure 1. First, the slack time t_{slack} is measured. If the slack time is zero, co-scheduled processes are preempted. Otherwise the co-scheduled processes are allowed to continue. When the slack time is positive, the interrupt timer is set at t_{next}.

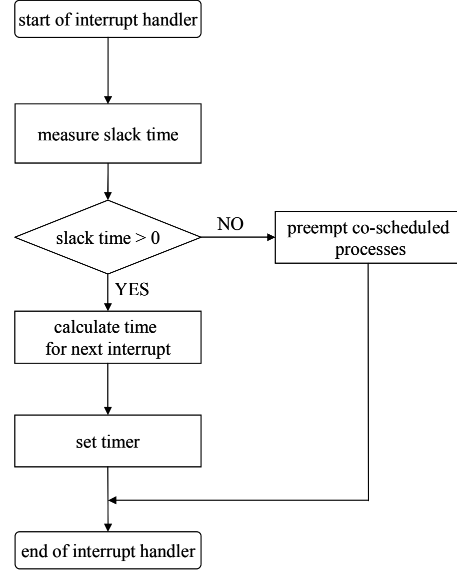


Figure 1. Flow chart of interrupt handler.

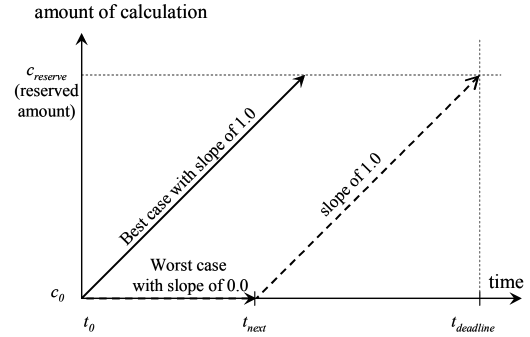


Figure 2. Schematic of calculation progress.

The other implementation issue is how to measure the slack time. To measure the slack time, we need a method to measure the amount of computation resources that a real-time process has already actually obtained. This measurement can be done in the either of the following ways:

1. A real-time process reports its progress by itself.
2. The task scheduler observes the progress by using a performance-monitoring facility of the processor.

The former requires modifying the real-time programs, while the latter requires a hardware assist. The strengths

and limitations of these two measurement methods are evaluated and discussed in Section 5.

Figure 2 illustrates a schematic image of the calculation progress of a real-time process. The x-axis shows the time and the y-axis shows the amount of computation resources that the real-time process has obtained up to that time. Because the amount of computation resources is represented by the single-threaded CPU time, the slope of the graph takes values from zero to one. The best case, with a slope of one, means no interference exists from co-scheduled threads, as in the single-threaded condition. On the other hand, in the worst case, the case with slope zero, the real-time process cannot use any computation resources and cannot make any progress because of the co-scheduled processes. Even in that worst case, however, since the co-scheduled processes are preempted at the time t_{next}, the real-time process can still meet its deadline. The time to monitor the slack time t_{next} presented by Equation (2) is determined on the basis of this worst case.

When using a fixed-interval polling timer to monitor the slack time, the basic approach is the same as for the variable interval timer. The fixed interval of the timer interrupt should be selected considering interrupt overhead and monitoring accuracy. If the interval is too small the overhead becomes significant, but if the interval is too long the monitoring becomes inaccurate, which means that the task scheduler cannot correctly detect when the slack time might become zero.

Our method trades off the overhead of external timer interrupts for a guarantee of the allocation of the reserved amount of computation resource to the real-time processes. When using a variable interval timer, if the minimum calculation speed of one thread in the SMT processor is given, the number of interrupts and overhead can be reduced using that minimum calculation speed in the calculation of t_{next} as

$$t_{next} = t_{slack} / (1 - \alpha) + t_0. \qquad (3)$$

where we define α as the ratio between the minimum speeds and the single-threaded calculation speeds (i.e. instructions per cycle). The value of α varies from zero to one. If α equals zero, the real-time process may be starved for computation resources due to competing co-scheduled threads. On the other hand, as α approaches one, the real-time process suffers less from the effects due to co-scheduled threads. For example, when α is 0.5, a real-time process is sure to run at a speed of at least half of the single-threaded speed and at single-threaded speed in the best case.

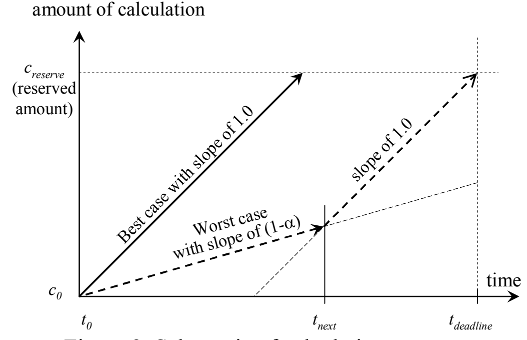


Figure 3. Schematic of calculation progress with reduction of timer interrupt.

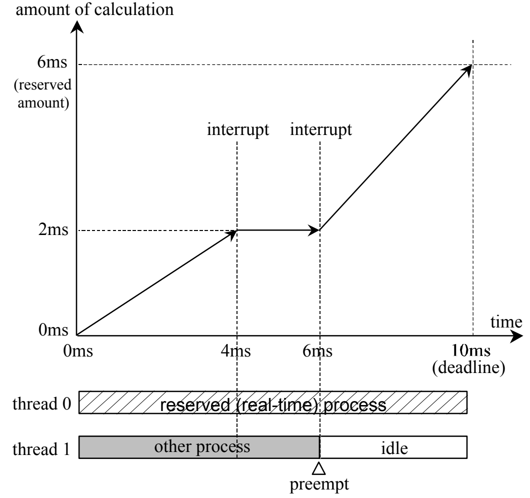


Figure 4. Sample case of scheduling (slack time monitoring method).

Figure 3 depicts a schematic image of the calculation progress of a real-time process using interrupt reduction. The y-axis of Figure 3 is the amount of computation resource represented by the single-threaded CPU time, as in Figure 2, so the slope of the graph in the best case is also one and it becomes α in the worst case. Even in such a worst case, it can be guaranteed that the real-time process can meet its deadline if the co-scheduled processes are preempted at the time of t_{next} calculated by Equation (3).

3.3 A scheduling example

Figure 4 shows a scheduling example. In this case, one real-time process and some other best-effort non-real-time processes are scheduled on a system with a two-thread SMT processor. The real-time process reserves computation resources corresponding to 6 ms of the single-threaded speed before the deadline at 10 ms. The slack time is monitored using a variable interval timer as described in the last section.

First, the real-time process is dispatched to thread 0. At that time slack time t_{slack} is calculated as,

$$t_{slack} = T - C = 10 \text{ ms} - 6 \text{ ms} = 4 \text{ ms}. \quad (4)$$

In this case t_{slack} is greater than zero; therefore one of the other best effort processes can be dispatched for thread 1. The next time to set the timer interrupt for is,

$$t_{next} = t_{slack} + 0 \text{ ms} = 4 \text{ ms}. \quad (5)$$

At the time 4 ms, the real-time process is observed to have already obtained the amount of computation resource corresponding to 2 ms of single-threaded speed, and thus the remaining reserved amount of computation resource is 4 ms. The slack time at this time is,

$$t_{slack} = T - C$$
$$= (10 \text{ ms} - 4 \text{ ms}) - (6 \text{ ms} - 2 \text{ ms}) = 2 \text{ ms}. \quad (6)$$

The slack time is still positive. Thus the co-scheduled best effort process is able to continue running on thread 1. The next time to set an interrupt is,

$$t_{next} = t_{slack} + 4 \text{ ms} = 6 \text{ ms}. \quad (7)$$

Between 4 ms and 6 ms, the real-time process cannot obtain any processor resources and the amount of computation resource already obtained is unchanged. Now the slack time becomes zero as follows:

$$t_{slack} = T - C$$
$$= (10 \text{ ms} - 6 \text{ ms}) - (6 \text{ ms} - 2 \text{ ms}) = 0. \quad (8)$$

Thus after this time, the real-time process must run alone on the processor to obtain the reserved amount of computation resource before the deadline. In order to run the real-time process in the single-threaded condition, the task scheduler preempts the co-scheduled best-effort process on thread 1, and makes thread 1 idle.

As described previously, disabling SMT while any real-time process is running is another way to guarantee the allocation of reserved computation resource. We call this naive method the *SMT disabling method*. Figure 5 depicts a sample case of scheduling using the SMT disabling method. In Figure 5 the real-time process is dispatched to thread 0 at time 0, and thread 1 remains idle from time 0. The real-time process can use all of the processor resources because thread 1 is idle, and can obtain the reserved amount of computation resources in 6 ms. In the remaining 4 ms, two of the best-effort processes can use thread 0 and thread 1. The SMT disabling method can guarantee real-time processes will meet their deadlines, though the SMT facility of the processor is not utilized while any

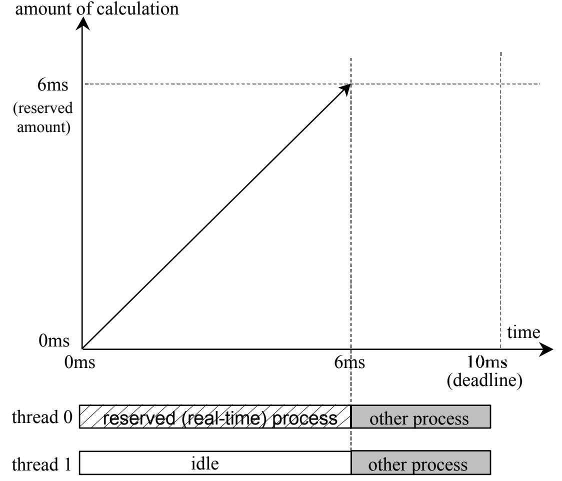


Figure 5. Sample case of scheduling (SMT disabling method).

real-time process is running, which usually cause the system to have lower throughput.

3.4 Scheduling of multiple real-time processes

We have described the basic scheduling scheme with only one real-time process in the last section. In this section, we consider a case with multiple real-time processes in the system. We denote the real-time processes as p_1, p_2,..., p_n and they are ordered by their deadlines. Thus process p_1 is the task with the earliest deadline and process p_n is the task with the latest deadline. Our proposed scheme, the slack time monitoring method, is intended to remove the effects of co-scheduled processes and does not define how to order the real-time processes to execute them. Existing scheduling schemes such as the earliest deadline first (EDF) scheduling or least slack-time first (LSF) scheduling can be used to define the order of real-time processes.

In the case with multiple real-time processes, the slack time for process p_i must account for the execution time of the processes p_1, p_2,..., p_{i-1}, which have the earlier deadlines. Thus the slack time for process p_i is calculated as follow,

$$t_{slack,i} = T_i - \sum_{j=1}^{i} C_j \quad (9)$$

where T_i is the remaining time to the deadline of the process p_i and C_i is remaining reserved amount of computation resource that process p_i must obtain before its deadline. The slack time for the overall system, which is used to determine whether co-scheduled

processes are allowed to continue or not, is the smallest one among the slack times for each real-time process.

4. Implementation

4.1 Basic Reservation Model

To evaluate the slack time monitoring method described in Section 3, we implemented a real-time task scheduler that uses this method in the Linux kernel 2.4.21 and evaluated it on a standard PC. The scheduler guarantees reserved amounts of CPU time for real-time processes. This reservation model is based on Resource Kernel [1, 2]. A CPU reservation used in this scheduler is specified by a reserved amount of computation resources represented by the single-threaded CPU time and a reservation period. Once a real-time process successfully creates a CPU reservation, the scheduler dispatches the real-time process at the highest priority until the real-time process uses up its reserved computation resource. If the real-time process uses up its reserved amount of computation resource or yields the CPU, the real-time process is not dispatched again until the end of current period. In a multi-processor or a SMT system, a CPU reservation additionally specifies which CPU the reservation uses. In this case the process with the reservation cannot use any other processors.

4.2 Scheduling algorithms for SMT processors

We have implemented three types of SMT scheduling algorithms. When the scheduler dispatches a real-time process with a reservation to one thread of the SMT processor, the scheduler handles other threads by using these three methods:

- *Non-SMT-aware*: The scheduler does not worry about SMT. This means that any other processes can run on other threads without any limitations.
- *SMT disabling method*: The scheduler dispatches real-time process only in the single-threaded condition. This means that no other processes can run on the SMT processor while a real-time process is running on it. This method is illustrated in Figure 5.
- *Slack time monitoring method (our proposed method)*: The scheduler monitors the slack time and dispatches other processes to other threads whenever possible, as described in Section 3.

Our implementation of the slack time monitoring method uses the system timer (IRQ 0) as a variable interval external timer. Thus the system timer is configured to generate one interrupt at a specified time, while it is configured to generate an interrupt periodically at the rate of the time slice quanta (10 ms) in the original Linux kernel. In order to measure the amount of computation that the real-time process has actually obtained, we implemented two methods in the scheduler.

1. *Syscall implementation:* A real-time process reports its progress by itself by using a system call.
2. *PMC implementation*: The task scheduler measures the progress by using the Performance Monitoring Counter (PMC) [14] of the Xeon processor.

We use the number of executed instructions measured by the PMC as a metric of the progress in the PMC implementation.

One more point was considered in this implementation to reduce the number of interrupts and overhead. We noted in Section 3 that co-scheduled processes are preempted when the slack time becomes zero. If the slack time becomes too small, the overhead becomes too large relative to the amount of time before the next interrupt. To avoid this, the co-scheduled processes are preempted when the slack time becomes less than 10 μs.

5. Evaluation

5.1 Experimental environment

In this paper, our proposed method was evaluated on a workstation with a Xeon processor, which is equipped with Intel's implementation of the SMT technique named Hyper-Threading. The single physical processor on this workstation was treated as two independent processors in the Linux kernel.

The evaluation reported here was done with only one real-time process using CPU reservations and two other best-effort processes of a non-real-time program. Several types of programs were tested as the best-effort process, and for each test two metrics were measured, deadline misses of the real-time process and throughput of the best-effort processes.

```
for (i=0; i<iSize; i++) {
    for (j=0; j<iSize; j++) {
        c[i][j] = 0.0;
        for (k=0; k<iSize; k++) {
            c[i][j] += a[k][j] * b[i][k];
        }
    }
    rt_report_progress(i*256/iSize);
}
```

Figure 6. Code of the real-time process.

5.2 Evaluation using simple workload

First we report results of an evaluation using a very simple workload. The real-time process used in this evaluation was a program that calculates the product of two 200×200 double-precision floating-point matrices once per time period. This program did not use the SIMD instructions or any special optimizations such as loop blocking. It was compiled using gcc 3.2 with the –O2 option. One matrix product calculation took about 50 ms on the workstation, and this calculation was done periodically with a CPU reservation with a period of 70 ms and a reserved computation resource corresponding to 55 ms. This meant that the real-time process should be able to process the matrix calculation at the highest priority until it obtained the computation resources corresponding to 55 ms in the single-threaded CPU speed. We confirmed that no deadline misses were observed under this condition on a non-SMT processor. Figure 6 shows the main code of the program used as the real-time process. In this code *rt_report_progress*() was the system call to report the progress of the computation to the scheduler. The argument of the system call was an integer value that describes the progress in the range between 0 and 255. This system call was used only when we evaluated the Syscall implementation and this line was commented out when we tested the PMC implementation.

5.2.1 Test for allocation of reserved amount of computation resource

First, whether or not the real-time process could obtain the reserved amount of computation resource was examined. Figure 7 depicts the completion times of the calculation in the presence of each one of three types of competing best-effort processes. The Syscall implementation of our proposed method is used in this evaluation. The x-axis is the time measured by the

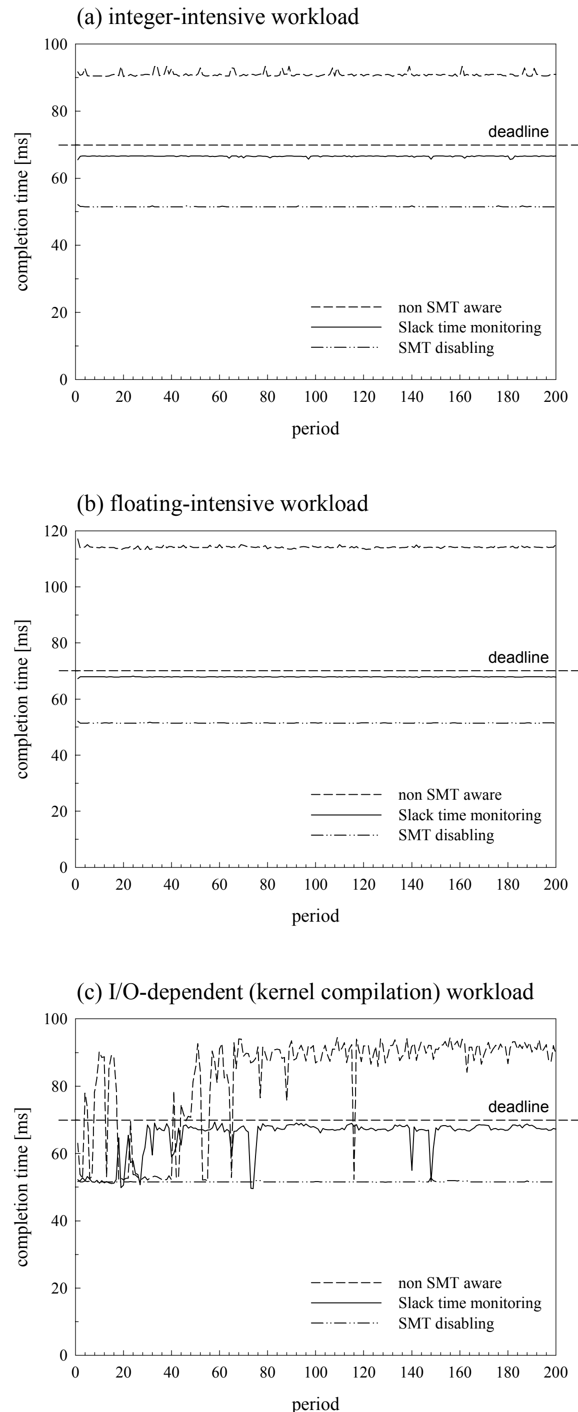


Figure 7. The completion times of the real-time process versus the number of iterations of periodical tasks for three competing tasks: (a) integer-intensive workload, (b) floating-intensive workload, and (c) I/O-dependent (kernel compilation) workload. (Slack time monitoring uses the Syscall implementation).

70-ms time periods, and the y-axis is completion times. If the completion time was smaller than the deadline time, 70 ms, it meant the real-time process successfully performed the reserved amount of computation before the deadline. If it was bigger than 70 ms, on the other hand, it meant the deadline was missed in that period. In Figure 7, three types of competing programs were used as best-effort processes. They were:

(a) a program that calculates the product of two 200×200 integer matrices

(b) a program that calculates the product of two 200×200 double-precision floating-point matrices (same as the real-time process)

(c) a program that compiles the Linux kernel repeatedly on the local hard disk drive (connected via an IDE interface and using Ext3 as the file system)

There were obvious differences between the methods in handling the other threads in the SMT processor. For the method which does not consider interference between the real-time process and a co-scheduled best-effort process, many deadline misses occurred regardless of the type of competing program. The real-time process was dispatched to a thread but a best-effort process was dispatched to another thread and it prevented the real-time process from obtaining the reserved amount of computation resources by competing for processor resources. In other words, the calculations of the real-time process became slower than in the single-threaded condition. This observation shows that the effect of co-scheduled processes must be considered in order to guarantee the allocation of the reserved amount of resources to a real-time process on an SMT processor.

For the SMT disabling method, the effects of the co-scheduled process on completion times were small enough to neglect and no deadlines were missed regardless of the type of competing program. This was because the other thread was idle while the real-time process was running, and there was no process to make the real-time process slow down by competing for processor resources, however, this also means that the performance benefit of SMT was lost. For the slack time monitoring method, it was also observed that deadline misses did not occur, though the completion times fluctuated due to the competing process. These results show that both SMT disabling method and slack time monitoring method can guarantee a real-time process will meet its deadline by allocating the reserved amount of computation resources to the real-time process.

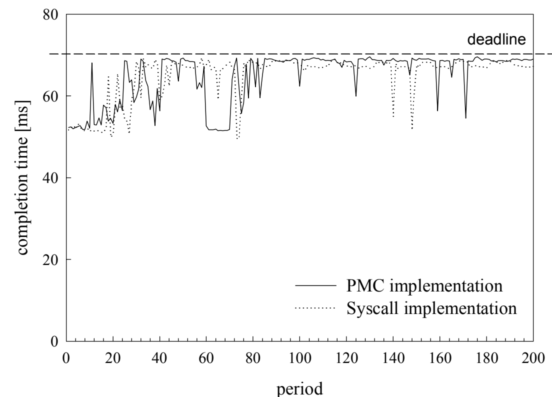


Figure 8. The completion times of real-time processes versus the number of iterations of periodical tasks for PMC and Syscall implementations.

Figure 8 depicts the result of the same evaluation using the PMC implementation of the slack time monitoring method. In this implementation the task scheduler measured the progress of the real-time process through the Performance Monitoring Counter and the real-time process did not need to report its progress. Figure 8 shows the result with the kernel compilation program as competing best-effort processes. The results of the PMC implementation shows there were no deadline misses though the completion times fluctuated. This result looks similar to that of the Syscall implementation in Figure 7. The results of the PMC implementation with other two types of competing programs are not shown here but they also similar to that of Figure 7. Thus both implementations of the slack time monitoring method worked well to guarantee a real-time process met its the deadline on an SMT processor.

5.2.2 Throughput of best effort processes

For the two methods that were able to guarantee allocation of the reserved amount of computation resources, the SMT disabling method and the slack time monitoring method, we evaluated the throughputs of the best-effort processes that were running while the real-time process was running with reservations. Figure 9 depicts the throughputs of the best-effort processes under the conditions described in the last section. The values are normalized using the results of the SMT disabling method. The throughput of the matrix calculation program was defined as the number of calculations done in unit time, and that of the kernel compile program was defined as the inverse of the required time to compile the entire kernel code as measured by the *time* command. We confirmed that the

real-time process did not miss its deadlines in these tests of throughput.

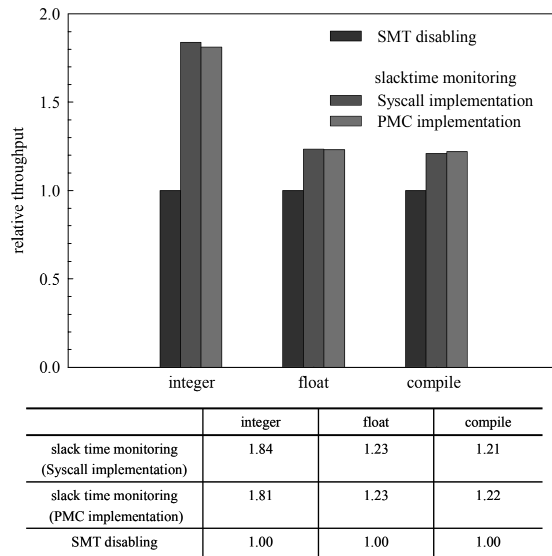
We can see that the slack time monitoring method led to higher throughput for the best-effort processes than the SMT disabling method in all of the programs and in both implementations. These results mean that the usage of processor resources is increased by the slack time monitoring method, because this method allows best-effort processes to run while the real-time process was running. The difference of throughput between both methods was not constant among the different types of best-effort processes. The most significant increase of throughput was observed for the integer matrix calculation, an increase of up to 84%, while a relatively small increase of 23% was observed for the floating-point matrix calculation. Because the real-time process used in our evaluation was a floating-point calculation intensive program, other best-effort floating-point calculation intensive programs could only obtain small increases in throughput by using SMT because of resource contention for the floating-point units. In the case of the kernel compile, which is thought to be a more practical job with disk I/O and process forks, the throughput was increased by about 20% by using the slack time monitoring method.

Comparing of the two types of implementations of the slack time monitoring method, the results showed that the differences between both implementations were small. This was because the overhead of the system call and that of an access to the performance monitoring counter were comparable.

Looking through all of these results, increases of throughput of the best-effort processes were observed for the slack time monitoring method comparing to the SMT disabling method, while both methods were able to guarantee allocation of the reserved amount of computation resources to the real-time process. The type of competing best-effort process affected the increase of throughput: More contention between processes led to less increase of throughput. However, the allocation of the reserved amounts of computation was not affected by the type of the best effort process, even if the process contained block I/O operations.

5.3 Evaluation using real workload

Here we evaluate the slack time monitoring method using more practical workloads that have time constraints. We choose a real-time MPEG encoding program as such a workload. In recent years, real-time MPEG encoding software is used in a wide range of



	integer	float	compile
slack time monitoring (Syscall implementation)	1.84	1.23	1.21
slack time monitoring (PMC implementation)	1.81	1.23	1.22
SMT disabling	1.00	1.00	1.00

Figure 9. The relative throughput of non real-time processes for three methods: SMT disabling method, Syscall implementation, and PMC implementation.

applications from modern PCs to consumer electronics devices such as set top boxes. A real-time MPEG encoder used with an external input such as a TV tuner must finish the encoding process of one frame before a the next frame comes, or the picture will be lost. Thus, a real-time MPEG encoder has strict deadline defined by the interval of picture frames. Though a buffer memory between the encoder and the tuner can help prevent such lost frames, there will still be losses when the buffer memory overflows. A real-time MPEG encoder is often used with other non-real-time processes in real systems. For example, a user may use a PC for Web browsing or other purpose while a TV program is being recorded in the background. An appropriate resource reservation system can help such encoder not to fail in the recording even though other heavy processes are running on the same machine.

In this evaluation, we used one of the most widely known open-source MPEG encoders *mpeg2enc* included in the MJPEG tools-1.6.2 [15]. The encoder was used to read an uncompressed video stream from a file on the local hard disk drive, encode it to MPEG-1 format, and write the encoded stream to the same disk. The input video stream was 30 seconds long and had a frame rate of 30 frames per second, so it contained 900 frames in total. The file size of the input data was about 100MB and the encoded data was 5.5MB. From this frame rate, the deadline for one encoding process was determined at 33.3 ms after the process started. The mpeg encoder can run in multi-threaded mode when

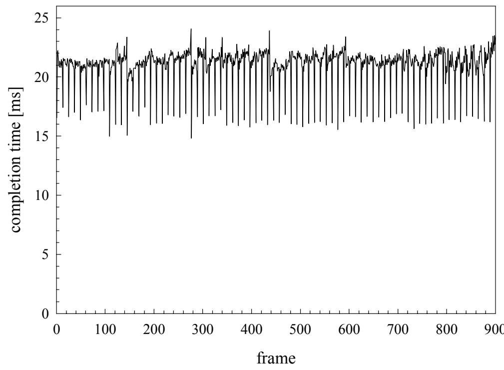


Figure 10. The completion times of MPEG encoding workload in each frame. (no competing tasks).

two or more processors are available, but we forced the encoder in single-thread mode in this test, whether or not Hyper-Threading was enabled.

First, the characteristics of the encoding process were tested. Figure 10 depicts the completion times of the encoding process of each frame without any competing processes. The completion times fluctuated even when no other processes affected them. One reason for this fluctuation was differences in the types of the frames. Two types of frames called I-frames and P-frames were included in the encoded MPEG stream; with one I-frame appearing after 11 P-frames. The computation time required to encode an I-frame is smaller than for a P-frame. Another reason of the fluctuation was changes of a scene in the video. There are some sharp peaks of the calculation time in Figure 10. These frames were drastically changed from the previous frames.

We have evaluated our implementations of the slack time monitoring method using this workload. In the test, the computation time of 25 ms was reserved for the encoder out of every 33.3 ms, and the deadline of the calculation was defined at 33.3 ms. For the Syscall implementation we had to modify the encoder program to insert the system calls to report its progress. We add the system calls at only two places, in an outermost large loop that does motion estimation and in another large loop that does a discrete cosine transformation. These parts consume a large portion of the entire computation time.

Figure 11 depicts the results of the completion times with the kernel compilation program running as the background competing process. The figure shows that both implementations of the slack time monitoring method and the SMT disabling method effectively prevented deadline misses, while the non-SMT-aware

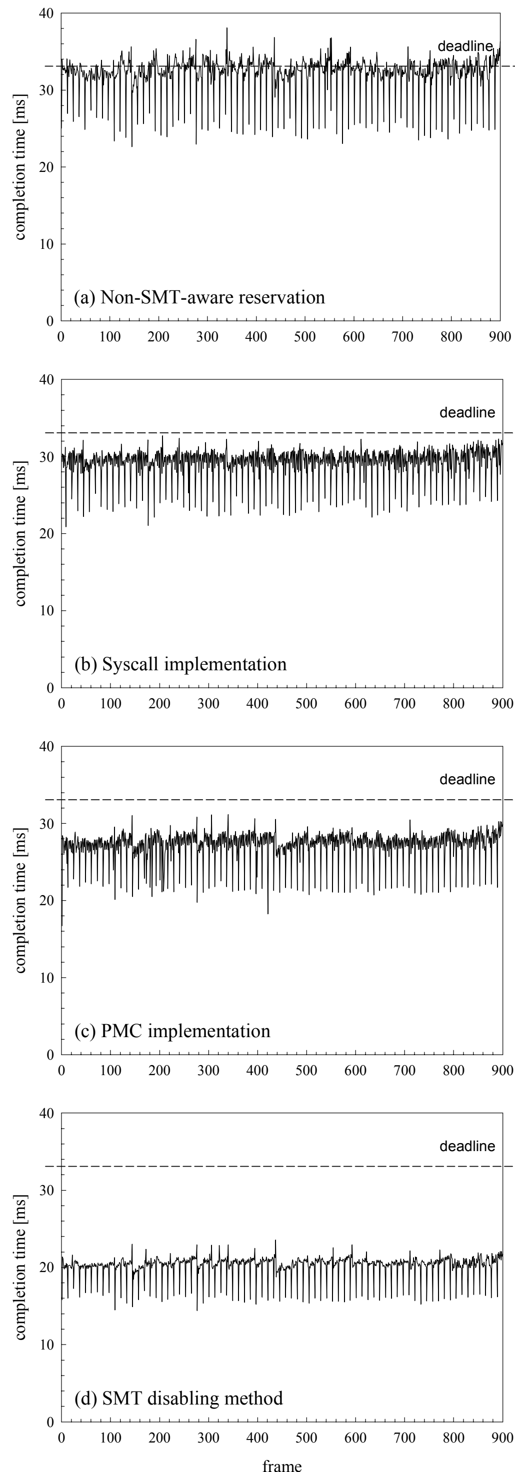


Figure 11. The completion times of MPEG encoding workload in each frame with a kernel compilation process as a competing job for four scheduling methods; (a) Non-SMT-aware reservation (b) Syscall implementation (c) PMC implementation (d) SMT disabling method.

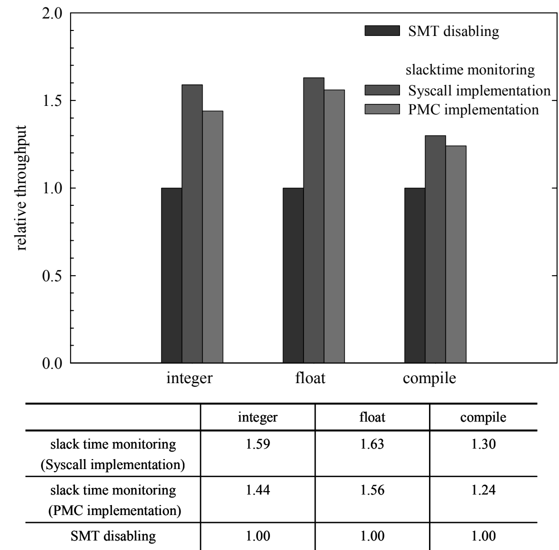
reservation method could not guarantee the encoder met its deadlines at all. The results of the Syscall implementation and the PMC implementation of the slack time monitoring method looked similar. Nevertheless we can see different tendencies in Figure 11. The completion times of the Syscall implementation tended to be later than that of the PMC implementation. This meant that the Syscall implementation could use the SMT facility more effectively and allow the non-real-time processes longer running times. Figure 10 shows that in this encoding workload the completion times of each frame fluctuated, and the degree of progress reported by the encoder itself can take into account such fluctuations of the required computation times, but the degree of progress measured by the task scheduler using the Performance Monitoring Counter could not do this. For example, if one frame in the video stream was very simple and the encoder required only 50% of the reserved computation resources to encode the frame, the task scheduler could not detect that the frame had not required the full computation resource and it had to conservatively dispatch all of reserved amount of resources to the encoder.

Figure 12 shows the throughputs of the best-effort processes that were running while the encoder was running as the real-time process with reservations. The throughputs of both implementations of the slack time monitoring method were better than the throughput of the SMT disabling method, just as with the simple workloads described in the last section. For the two implementations of the slack time monitoring method, the Syscall implementation obtained slightly higher throughput than the PMC implementation. This was consistent with the results shown in Figure 11.

From these results, we see that the Syscall implementation has more opportunities to utilize the SMT facility through it requires program modifications that the PMC implementation does not require.

6. Future Work

In this paper, we evaluated only one real-time process as the first step toward using an SMT processor for a real-time application that requires deterministic allocation of processor resource. More work is needed to use SMT processors for real-time applications in general conditions. Some of the most important possible future works would be considering two or more real-time applications running on a system, SMT processors with more than two hardware threads and



	integer	float	compile
slack time monitoring (Syscall implementation)	1.59	1.63	1.30
slack time monitoring (PMC implementation)	1.44	1.56	1.24
SMT disabling	1.00	1.00	1.00

Figure 12. The relative throughput of non real-time processes for three methods: SMT disabling method, Syscall implementation, and PMC implementation.

systems with multiple SMT processors. The SMT facility of IBM POWER 5 processor can assign different priority level to each thread [16]. To utilize this mechanism in our method will be another challenging future work.

7. Conclusion

This paper proposes a novel scheduling method to exploit SMT facilities in a system that contains real-time and best-effort processes. On SMT-enabled systems, existing reservation-based scheduling methods may fail to allocate resources that real-time applications require, and hence real-time applications may fail to meet their computation deadlines. This is because existing reservation-based methods assume the CPU time of a process indicates the amount of the resource that the process has consumed and because the assumption is no longer valid on SMT processors. Our slack time monitoring method, on the other hand, keeps track of the progress of real-time processes and adaptively enables/disables the SMT facility. In this way, our method can achieve a high throughput by exploiting the SMT facility even when the workload contains both real-time and best-effort processes. To our best knowledge, this paper is the first report that quantitatively evaluated reservation-based real-time

scheduling schemes on a commercially-available SMT environment.

We have implemented the slack time monitoring method on a Linux kernel and have evaluated the system throughput on a Hyper-Threading processor, which is an Intel's implementation of the SMT technique. Our experimental results have shown that our method can always allocate resources that our real-time applications have previously reserved, while a traditional reservation-based method that does not aware SMT fails to do so when the SMT facility is enabled. Yet, our method can still exploit the SMT facility; our method results in up to 84% higher throughput of best-effort processes than a scheduling method that blindly disable SMT facility while real-time process is running.

We evaluated the following two implementations of our slack time monitoring method. The first implementation relies on the task scheduler to monitor the progress of real-time processes by using the Performance Monitoring Counter. The second implementation relies on each real-time application to report its progress to the task scheduler. Our results have shown that the second one can produce a higher throughput than the first one. This is because the second implementation allows the task scheduler to dispatch best-effort processes longer than the first implementation by using run-time information while the first implementation requires the task scheduler to assume the worst-case resource usage.

We expect that SMT techniques are becoming popular not only for servers and workstations but also for embedded systems. Since embedded systems often involve real-time applications, it is becoming increasingly important to exploit SMT facilities for real-time systems.

References

[1] R. Rajkumar, K. Juvva, A. Molano, S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time Systems, *In Proceedings of the Conference on Multimedia Computing and Networking*, SPIE/ACM (1998).

[2] S. Oikawa, R. Rajkumar. Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior, *In Proceedings of the Real-Time Technology and Applications Symposium*, IEEE (1999).

[3] M. B. Jones, D. Rosu, M. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities, *In Proceedings of the Symposium on Operating Systems Principles*, ACM (1997).

[4] C. W. Mercer, S. Savage, H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. *In Proceedings of the International Conference on Multimedia Computing and Systems*, IEEE (1994).

[5] C. L. Liu, J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *Journal of the ACM*, Vol. **20**, No. 1, pp. 46–61 (1973).

[6] D. Tullsen, S. Eggers, H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism, *In Proceedings of the Annual International Symposium on Computer Architecture*, ACM (1995)

[7] A. Snavely, D. Tullsen. Symbiotic Job scheduling for a Simultaneous Multithreading Processor, *In Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ACM (2000).

[8] A. Snavely, D. Tullsen, G. Voelker. Symbiotic Job scheduling with Priorities for a Simultaneous Multithreading Processor, *In proceedings of the SIGMETRICS international conference on Measurement and modeling of computer systems*, ACM (2002).

[9] S. Parekh, S. Eggers, H. Levy. Thread-Sensitive Scheduling for SMT Processors, *University of Washington Technical Report* (2000).

[10] J. Nakajima, V. Pallipadi. Enhancements for Hyper-Threading Technology in the Operating System - Seeking the Optimal Scheduling, *In Proceedings of the Workshop on Industrial Experiences with System Software*, USENIX (2002)

[11] R. Jain, C. J. Hughes, S. V. Adve. Soft Real-Time Scheduling on Simultaneous Multithreading Processors, *In Proceedings of the Real-Time Systems Symposium*, IEEE (2002).

[12] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, M. Valero. Predictable performance in SMT processors, *In Proceedings of the conference on Computing Frontiers*, ACM (2004)

[13] Intel Corporation. Hyper-Threading Technology, *Intel Technology Journal*, Vol **6**, No 1 (2002)

[14] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide.

[15] MJPEG Tools. http://mjpeg.sourceforge.net/

[16] R. Kalla, B. Sinharoy, J. M. Tendler. IBM power5 chip: a dual-core multithreaded processor, *IEEE Micro*, Vol. **24**, No. 2, pp. 40-47 (2004).

TM Xeon is a trademark of Intel Corporation.
POWER is a trademark of International Business Machines Corporation.