

Overhead-Aware Schedulability Analysis on Responsive Multithreaded Processor

Yusuke Kumura, Keigo Mizotani, Masayoshi Takasu, Hiroyuki Chishiro, and Nobuyuki Yamasaki
Graduate School of Science and Technology, Keio University
3-14-1, Hiyoshi, Kohoku-ku Yokohama, Kanagawa, 223-8522, Japan
(yusuke, mizotani, takasu, chishiro, yamasaki)@ny.ics.keio.ac.jp

Abstract

In recent real-time systems, having multicore or simultaneous multithreaded (SMT) processor is now widespread. Considering the complexity of today's architecture, predicting the behavior of task execution is challenging. And this has become a severe problem in real-time systems because estimating worst-case execution time (WCET) of tasks is important in constructing a dependable and accurate real-time systems. In this research, we provide overhead-aware schedulability analysis on a SMT processor. We take measurement-based approach in order to estimate WCET of tasks by measuring various runtime overheads that a real-time OS poses, because runtime overheads have a considerable impact on systems. Those measured overheads should be incorporated in schedulability analysis to evaluate the system more accurately. We target Dependable Responsive Multithreaded Processor (D-RMTP), an 8-way SMT processor, to estimate runtime overheads, and conduct a series of schedulability analysis with measured runtime overheads considered. Our evaluation shows that the advantage of having context cache mechanism, the unique functionality of D-RMTP, can improve schedulability by up to 15.9%.

1 Introduction

Today, a large part of our life is supported by information systems. Especially, real-time systems such as automobiles, airplanes and humanoid robots attract huge social attention due to its critical characteristics. In real-time systems, correctness of processing tasks depends not only on its result but also on time. Therefore, the system as a whole needs to meet timing constraints at the same time as processing correctly. Since complicated systems such as humanoid robots have become quite widespread recently, new kinds of requisite features have appeared such as processing computationally hard tasks. As the demand of system throughput requirement increases, we faced perfor-

mance limitation with only one execution context in the system. In order to tackle this problem, the system requires more than one processor or processing context: multicore and simultaneous multithreading (SMT) [9].

In addition, determining worst-case execution time (WCET) of running tasks is important in constructing an accurate real-time system. WCET should be obtained in advance by analyzing each task. Theoretically, it is possible to analyze WCET correctly by considering all related states and conditions, for example cache states and processor pipeline. Recently, there are some methods of WCET analysis for uniprocessors, for example commercial WCET analysis tools. However, these methods are not available and exceptionally challenging for state-of-the-art complex computer architectures: multicore and SMT processor because of the underlying architecture's components, such as caches and branch predictor. For SMT processors, it is still not tractable with existing WCET analysis tools as well [10]. Therefore, there is currently no option but to employ empirical methods. We measured runtime overheads in the OS development phase using a measurement-based approach [2] in order to estimate WCET of tasks, because runtime overheads cannot be neglected in practice.

In the research by Calandrino et al. using a real-time patch for Linux LITMUS^{RT} [5], authors evaluated multicore real-time scheduling algorithms (Partitioned EDF, Global EDF and Pfair [1]) with run-time overhead considered. In the research by Brandenburg et al. [3], authors showed that OS implementation aspects of schedulers, such as the design of data structures for queue and interrupt handler, have a strong impact on schedulability, indicating run-time overhead affects the choice of schedulers. In addition, in [4], authors introduced techniques to incorporate run-time overheads into schedulability analysis. In this research, we employ the preemption-centric interrupt accounting method, which is briefly introduced in the next section. In the research by Gracioli et al. [6], authors compared their RTOS called EPOS with

LITMUS^{RT}, and showed that properly designed RTOS can provide hard real-time guarantee close to theory, and how different OSs have impact on hard real-time schedulers.

To the best of our knowledge, this research is the first to conduct overhead-aware schedulability analysis on an 8-way SMT architecture. In this research, our target architecture is Responsive Multithreaded Processor (RMTP) [11] which is designed and developed for processing tasks on distributed real-time systems. We used a measurement-based approach to conduct schedulability analysis with real runtime overheads considered.

The remainder of this paper is organized as follows. The next section explains the model of this research, including the task model, runtime overhead, and overhead accounting method. Section 3 explains the characteristics of our target processor RMTP and its functionality, and OS implementation. In Section 4, we first explain overhead measurement and conduct a series of schedulability analysis. Finally, we conclude with a summary and future work in Section 5.

2 Task Model and Overhead Accounting Method

2.1 Task model

In this research, we consider all tasks to be periodic. In a task set τ , it has n tasks $\{T_1, T_2, T_3, \dots, T_n\}$. Each task T_i has a period p_i and worst-case execution time e_i . And the utilization is defined as the fraction of the worst-case execution time in the period. Each task T_i releases a job at every period p_i . And we consider implicit deadline model where each job's deadline is the next release time of the same task. The relative deadline of task T_i is equal to its period p_i . These n tasks are scheduled on m contexts $\{C_1, C_2, C_3, \dots, C_m\}$.

2.2 Scheduling Algorithms

In this research, we consider two real-time scheduling algorithms: Global-EDF (G-EDF) and Partitioned-EDF (P-EDF). In earliest-deadline first (EDF) scheduling, jobs are scheduled in the order of urgent deadline, giving higher priority to a job with closer deadline. EDF is known to be optimal uniprocessor real-time scheduling algorithm. G-EDF and P-EDF are scheduling algorithm for multi context systems. G-EDF algorithm manages all jobs on a single unified queue, and jobs are sorted by deadlines. And a scheduler picks jobs in ascending order of deadline. A job can be scheduled on available contexts and can also be

migrated to other context online. In addition, each job can be preempted by other highly prioritized job. P-EDF algorithm assigns tasks to each context offline, and each task is not allowed to be migrated to other contexts. All jobs are managed on a queue for its assigned context. On each queue, tasks are scheduled under EDF scheduling algorithm.

2.3 Scheduler Overhead

In this section, we list the types of scheduler overheads. Figure 1 depicts various runtime overheads affecting task execution, and the task set can miss a deadline due to the overheads. We consider 6

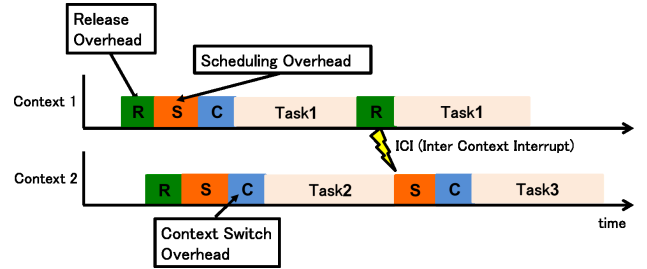


Figure 1: Example of runtime overhead

runtime overheads, and Table 1 lists the overheads. The context switch overhead is the time spent in the process of saving and restoring the context of currently running task and the next running task. The scheduling overhead is the time consumed to pick a next task to run. This process includes traversing and removing the picked task from the ready queue and changing the state of the task. The tick overhead is the time of timer interruption handler. This process includes updating tick count, i.e. jiffies. On every tick, the tick overhead is consumed. The release overhead is the time spent to release a task. This process includes choosing tasks that reached its release time from wait queue and inserting them into the ready queue, and changing the state of tasks. The inter context interrupt overhead is the time spent to notify other context to run scheduler, for rescheduling a job on a different context. The cache overhead is the cost of cache system, such as reestablishing cache affinity after preemption, migration, and context switch. With regard to the cache overhead, we do not consider it in this research.

2.4 Overhead Accounting

There are techniques to incorporate various scheduler overheads in schedulability analysis proposed in the literature. We use the preemption-centric interrupt accounting [2], which transforms the execution time of

Table 1: List of scheduler overheads

Notation	Overhead Type	Description
Δ^{cs}	Context Switch Overhead	Time required to switch between two contexts
Δ^{sch}	Scheduling Overhead	Time to pick a next running task
Δ^{tck}	Tick Overhead	Time required to handle periodic timer interrupt
Δ^{rel}	Release Overhead	Time to release all jobs that have reached their release time
Δ^{ici}	Inter Context Interrupt Overhead	Time to notify other context by issuing interruption
Δ^{cpd}	Cache Overhead	Cache related preemption and migration delay

task T_i to account for scheduler overheads.

$$e_i' = \frac{e_i + 2 \cdot (\Delta^{sch} + \Delta^{cs}) + \Delta^{cpd}}{1 - u_0^{tck}} + 2 \cdot c^{pre} + \Delta^{ici} \quad (1)$$

Where:

Q is the period of a tick interrupt (Quantum)

$u_0^{tck} = \frac{\Delta^{tck} + \Delta^{rel}}{Q}$ (fraction of tick and release overhead in a quantum)

$c^{pre} = \frac{\Delta^{tck} + \Delta^{rel}}{1 - u_0^{tck}}$ (the cost of one preemption)

Equation (1) transforms the original execution time e_i of task T_i into inflated execution time e_i' accounting various runtime overheads: context switch, scheduling, tick, release, inter context interrupt, and cache overhead. Using the technique, execution time of each generated task is inflated and used in the schedulability analysis explained later.

3 RMTP and Implementation

In this research, our target architecture is Responsive Multithreaded Processor (RMTP) [11]. We have been developing RMTP for distributed real-time systems, especially for humanoid robots. RMTP is an 8-way SMT processor, with up to 8 threads can be executed simultaneously. In this section, we explain the core functionality of RMTP and scheduler implementation.

3.1 Context Cache

The context cache [11] is an on-chip memory dedicated to saving hardware contexts. In general, context switch requires saving and restoring hardware contexts by software. Software context switch includes saving and restoring all general purpose registers, floating point registers, and control registers, consuming hundreds of thousands of clock cycles. Figure 2 shows the context cache mechanism implemented on RMTP. On RMTP, it has an exclusive bus for data transfer making context switch possible in 4 clock cycles. RMTP is an 8-way SMT processor, which has 8 active threads

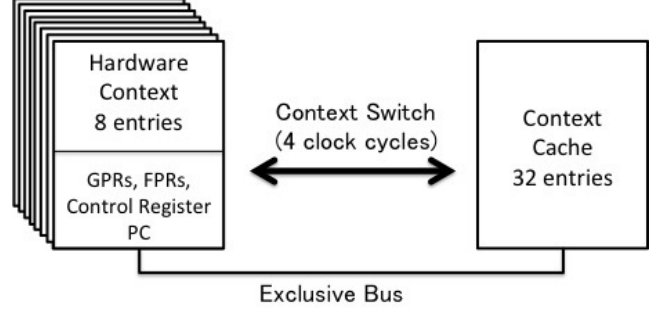


Figure 2: Context Cache

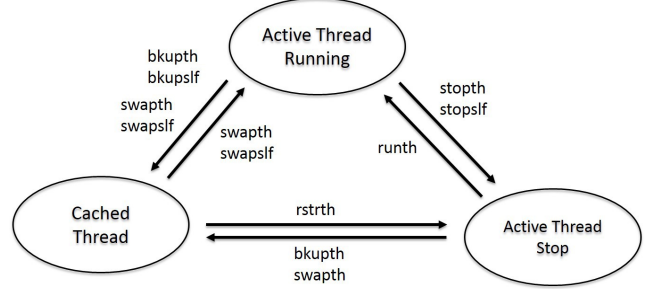


Figure 3: State transition chart of RMTP threads

running simultaneously and 32 cached threads stored in the context cache.

3.2 RMTP Thread Instructions

Figure 3 shows the state transition of RMTP threads. An executable thread is called active thread, and other 32 threads stored in the context cache is called cached thread. Therefore, 40 threads in total can be used without saving and restoring all the register values in main memory.

Each transition is triggered by software instructions. Table 2 shows the thread control instructions of RMTP. These instructions are used in OS, such as context switching, creating and deleting tasks, running and stopping tasks. The general idea of context switch implementation using the context cache is shown in Algorithm 1. It is possible to switch threads on RMTP

Table 2: List of RMTP thread control instructions

Instruction	Description
mkth	Create new thread
delth	Delete specified thread
runth	Set specified active thread running
stopth	Set specified active thread stopped
stopslf	Set calling active thread stopped
bkupth	Store specified thread to context cache
bkupslf	Store calling thread to context cache
rstrth	Set cached thread to active thread
swapth	Swap active thread and cached thread
swapslf	Swap calling thread and cached thread

by only issuing RMTP thread swap instruction if the switching target thread is a cached thread. In this research, since we target embedded real-time systems, we only consider the system with up to 40 tasks, which is enough for embedded real-time systems. Therefore, saving and restoring thread information to and from memory do not occur, and all the context switch take place only with thread control instructions of RMTP.

Algorithm 1 Context Switch on RMTP

Require:

```

1:  $A \leftarrow \{A_1, \dots, A_8\}$   $\triangleright$  Active Threads
2:  $C \leftarrow \{C_1, \dots, C_{32}\}$   $\triangleright$  Cached Threads
3: Let  $prev$  be the currently running thread on the context
4: Let  $next$  be the next running thread on the context
5:  $prev$  and  $next$  are already selected by the scheduler
6:  $prev \in A$ 
7: Total number of tasks is equal to or less than 40
8:
9: function RMT_THREADSWAP( $prev, next$ )  $\triangleright$  context switch  $prev$  and  $next$ 
10:   if  $next \in C$  then
11:     swapth  $prev$  and  $next$ 
12:   end if
13: end function

```

3.3 Other Implementation Issues

We implemented P-EDF and G-EDF algorithms. For both algorithms, we used priority queue as ready queue. Therefore, scheduling overhead does not include traversing ready queue because it picks next task by extracting the first element. And task releasing code handles all the work required to maintain a ready queue to be a priority queue. In order to record each runtime overhead, overhead tracing code is embedded at the start and end of each overhead generating code.

Overhead tracing codes read current clock cycle counter and store it on memory, and using the subtraction of the start from end as the overhead. The system tick is set to 10 ms. At every tick, every timer interrupt triggers tick overhead and release overhead.

4 Evaluation

In this section, we first explain the overhead measurement. Next, we explain the schedulability analysis using the obtained overheads.

4.1 Overhead Measurement Environment

We targeted Dependable Responsive Multithreaded Processor (D-RMTP) [8], that is one version of RMTP. We used evaluation kit which has System in a Package (SiP) with D-RMTP System on a Chip (SoC), to collect various overheads data under various task set scenarios. Table 3 shows the hardware specification of D-RMTP. Also, it has 32KBytes of instruction and data cache, 64KBytes of SRAM and 64MBytes of SDRAM. All the programs were executed entirely on SRAM.

Table 3: Hardware Specification of D-RMTP

Clock Frequency	50MHz
Active Threads	8
Cached Threads	32
Integer Register	32bit $\times$ 32 entry $\times$ 8-set
Floating Point Register	32bit $\times$ 64 entry
ALU	4 + 1 (divider)
FPU	2 + 1 (divider)
Branch Unit	2
Memory Access Unit	1

4.2 Task set Generation

We generated random task sets in order to conduct the overhead measurement and the schedulability analysis. We used three distributions of period and utilization. These distributions are the same as parameters used in the research by Brandenburg [2], generating a total of 9 scenarios. Task utilizations include three uniform distributions. The ranges are [0.001, 0.1] (light), [0.1, 0.4] (medium), and [0.5, 0.9] (heavy). The ranges of task period distributions are [3ms, 33ms] (short), [10ms, 100ms] (moderate), and [50ms, 250ms] (long). We employed Worst-Fit algorithm for partitioning generated task set in this research.

4.3 Overhead Result

We measured each type of runtime overheads, tick, release, scheduling, and context switch overheads, by running the overhead measurement programs on D-RMTP evaluation kit. For each combination of scheduling algorithm and the number of hardware contexts, we ran the overhead measurement program changing the number of tasks, and collected a number of overhead records. We set the number of hardware contexts to 2, 4, and 7. And the maximum number of tasks in a task set is set to an addition of the number of hardware contexts and 32 (the number of cached threads). We measured 2 types of context switch overhead. One measured using RMTP thread instructions to switch between threads, and the other one measured by doing general context switch procedures, i.e., saving and restoring all necessary registers by software. We generated task sets and executed them on the evaluation kit, changing the number of threads and tasks. We evaluate both average and worst case values.

4.3.1 Tick Overhead

Figure 4 shows the average and worst case tick overheads for P-EDF, and Figure 5 shows the results for G-EDF. The cost is almost constant for both algorithms, in range of 1-5 us. This is because the tick handler code is the same between P-EDF and G-EDF. Since the contention of hardware resources become intense as the number of contexts increases on a SMT processor, the tick overheads of 7 contexts is the largest in most cases.

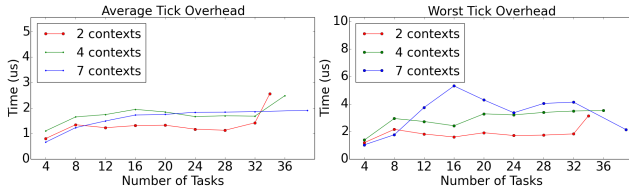


Figure 4: Tick Overhead (P-EDF)

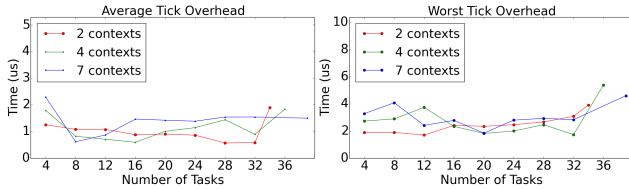


Figure 5: Tick Overhead (G-EDF)

4.3.2 Release Overhead

Figure 6 shows the average and worst case release overheads for P-EDF, and Figure 7 shows the results for G-EDF. With regard to the release overheads, the cost is dependent to the number of tasks in the system for both P-EDF and G-EDF because the number of tasks to be moved to the ready queue increases in a given tick. The cost for G-EDF is larger than P-EDF in most cases because of the design of task release handling code. Since G-EDF manages only one global queue, it handles more tasks than P-EDF per queue. However, the cost of P-EDF did not become as small as the algorithm implies. We think that this is because of the contention of hardware resources, which is the characteristic feature of a SMT processor, making the cost larger than we expected.

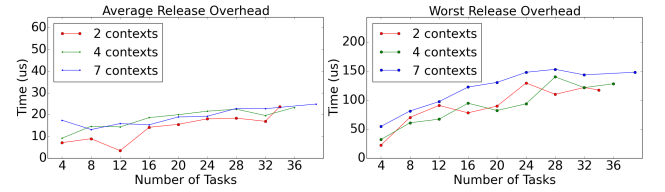


Figure 6: Release Overhead (P-EDF)

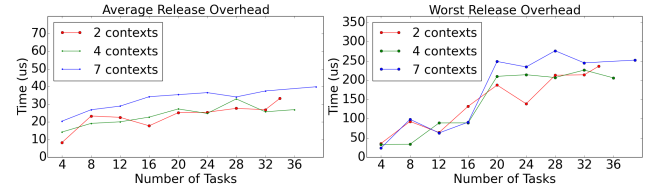


Figure 7: Release Overhead (G-EDF)

4.3.3 Scheduling Overhead

Figure 8 shows the average and worst case scheduling overheads for P-EDF, and Figure 9 shows the results for G-EDF. For P-EDF, since the scheduler has to only pick the next running task from the head of ready queue. Therefore, the cost is smaller than G-EDF. On the other hand, G-EDF has to handle all the contexts in the system, such as running new tasks on idle threads or swapping higher priority task with currently running lower priority task, i.e., preemption. Algorithmically, the size is dependent to the number of contexts and tasks.

4.3.4 Context Switch Overhead

Figure 10 shows the average and worst case context switch overheads for P-EDF, and Figure 11 shows the results for G-EDF. This is the overhead of using

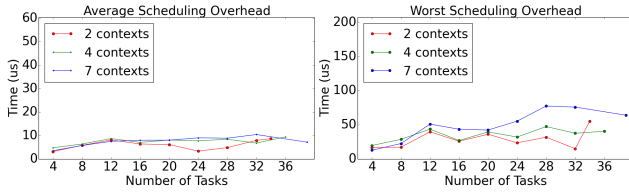


Figure 8: Scheduling Overhead (P-EDF)

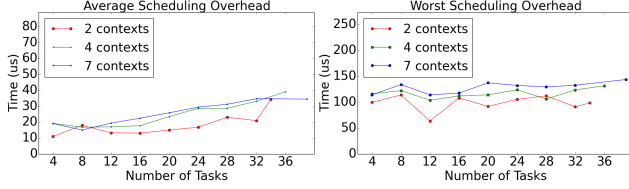


Figure 9: Scheduling Overhead (G-EDF)

context cache mechanism of D-RMTP. For context switch overhead, the cost is almost constant for both algorithms, in the range of 5-15 us. This is because the context switch code is same between P-EDF and G-EDF.

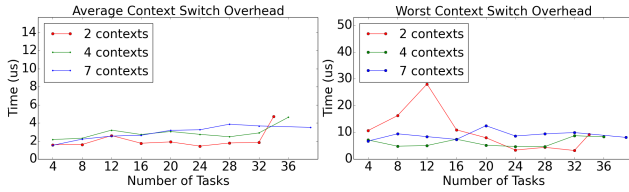


Figure 10: Context Switch Overhead (P-EDF)

4.3.5 Other Overheads

In order to compare the effect of context cache functionality of D-RMTP, we also measured the overhead of software context switching, which requires saving and restoring all the registers including integer, floating point registers and control registers through memory. In the worst case, since every memory access (instruction fetch and load/store operations) introduces a cache miss, we measured the overhead disabling a cache system. The software context switching overhead is 1,384 clock cycles in total.

For inter context interrupt overhead, we measured in the same way with software context switching. And the overhead is 644 clock cycles in total.

We incorporated all overheads shown above in generated task sets for the schedulability analysis. For tick, release, scheduling, and context switch overhead, we used the worst measured value for generated task sets in order to estimate the worst case scenario. When the number of tasks in the generated task set does not match the parameter used for overhead measuring, we

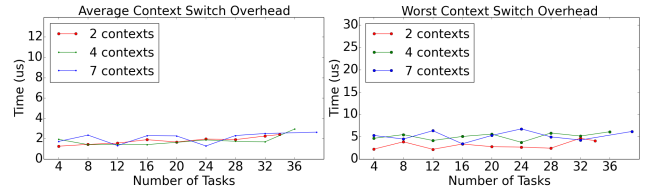


Figure 11: Context Switch Overhead (G-EDF)

used linear interpolated value between the two closest number of tasks used in the overhead measuring.

For software context switching and inter context interrupt overhead, we used the measured constant value for all task sets.

4.4 Schedulability Evaluation

We conducted a series of schedulability analysis using the overheads obtained in the overhead measurement phase. The schedulability analysis consists of two parts: schedulability test and schedulability simulation.

4.4.1 Schedulability Test

Schedulability test is an offline test that can judge whether a given task set is schedulable or not under a given scheduling algorithm. Therefore, being able to pass the test means the given task set is schedulable. However, since being able to pass the test is a sufficient condition for the task being schedulable, the result can be pessimistic.

With regard to the schedulability test for P-EDF, a task set is first mapped to each per context task queue, and check whether the utilization of all per context task queue does not exceed 100%. If any of the utilization of per context task queue exceeds 100%, the task set is judged to be unschedulable.

There are some schedulability tests for G-EDF proposed in the literature, we used 4 tests, Baker's test, GFB test, BAR test, and RTA test [7]. And a task set is judged to be schedulable if it passes any of the 4 tests. For each combination of algorithm, task period distribution, task utilization distribution and number of tasks, we generated 1,000 task sets and conducted the schedulability analysis changing the number of contexts on the platform. Schedulability is defined as the number of schedulable task sets out of all tested task sets, i.e., 1,000 task sets.

Figure 12 and Figure 13 shows the schedulability test results for P-EDF and G-EDF with task set of uniform light utilization and uniform short period running on 2 contexts. Schedulability improved 15.9% at maximum for P-EDF and 11.4% for G-EDF compared to the results of not using context cache mechanism. Figure 14

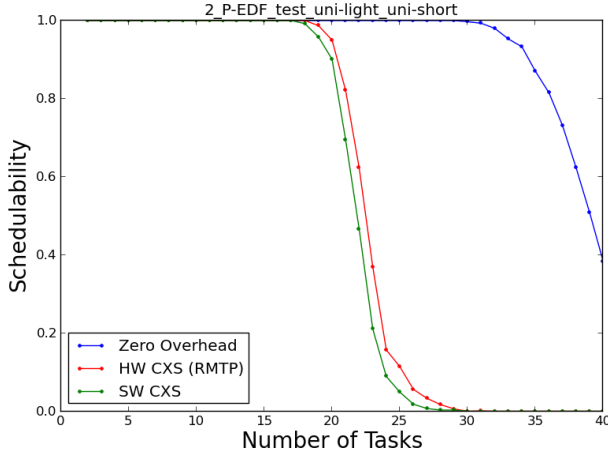


Figure 12: Schedulability Test on P-EDF (2 contexts)

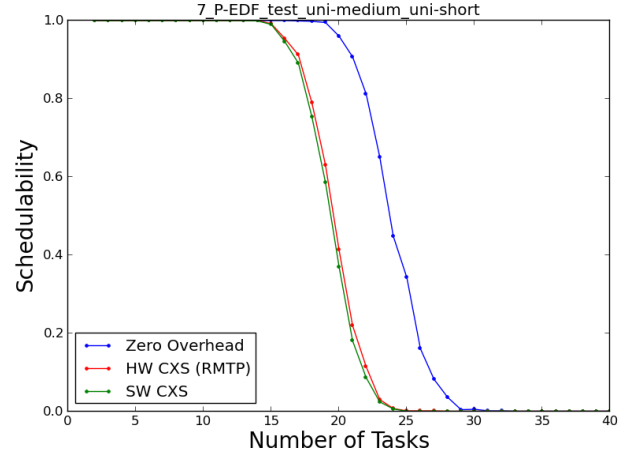


Figure 14: Schedulability Test on P-EDF (7 contexts)

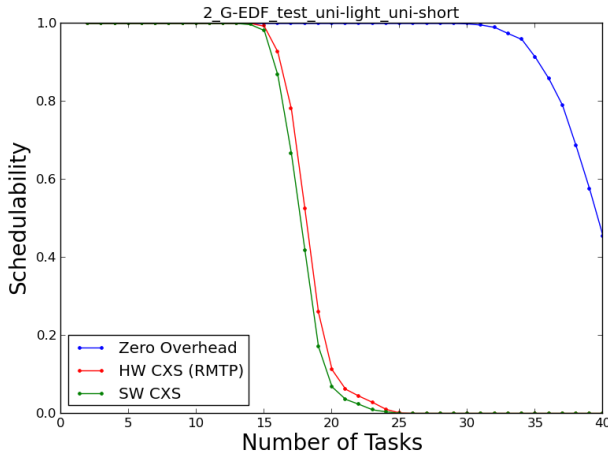


Figure 13: Schedulability Test on G-EDF (2 contexts)

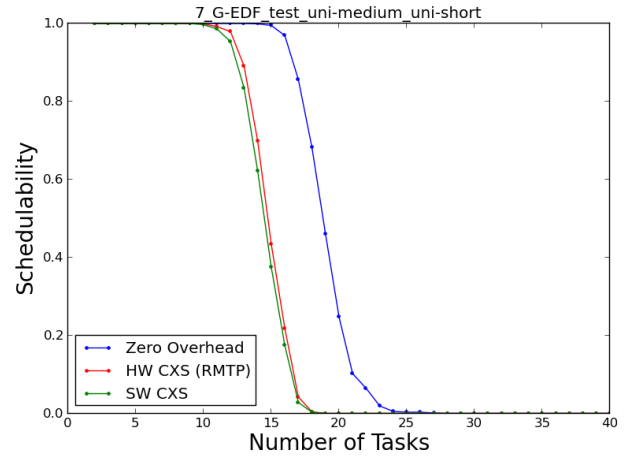


Figure 15: Schedulability Test on G-EDF (7 contexts)

and Figure 15 shows the schedulability test results for P-EDF and G-EDF with task set of uniform medium utilization and uniform short period running on 7 contexts. Schedulability improved 4.5% for P-EDF and 7.7% at maximum for G-EDF using context cache mechanism. We omit other evaluation in different scenarios, which show similar results, due to a limited number of pages. It is clear that considering runtime overheads degrades schedulability substantially making theoretically schedulable task set unschedulable. For example, in Figure 12, when the number of tasks is 30, none of the all task sets are schedulable considering the overhead while all the task sets are schedulable if all the overheads are ignored. The similar thing can happen on other architectures. And the overhead-aware schedulability analysis can help building more accurate real-time systems. In addition, using context cache mechanism definitely improves schedulability reducing the context switch overhead.

4.4.2 Schedulability Simulation

Schedulability test is a lightweight tool to evaluate schedulability, however, it can be pessimistic because a given task set that was judged to be unschedulable might be schedulable if actually be executed on the platform. In this sense, it is also meaningful to conduct schedulability simulation that simulates actual execution of a given task set. We used exactly the same generated task sets used in the schedulability test, and the total simulation length is set to 10 seconds. Figure 16 shows the schedulability simulation result for G-EDF with task set of uniform medium utilization and uniform short period running on 7 contexts. Compared to the result of schedulability test with exactly the same parameters: generated task sets, a scheduling algorithm and a number of contexts (Figure 15), schedulability improved making more task set to be schedulable. Similar to the evaluation of schedulability test, using context cache improves up to 5.6%. Similar results were obtained in other scenarios.

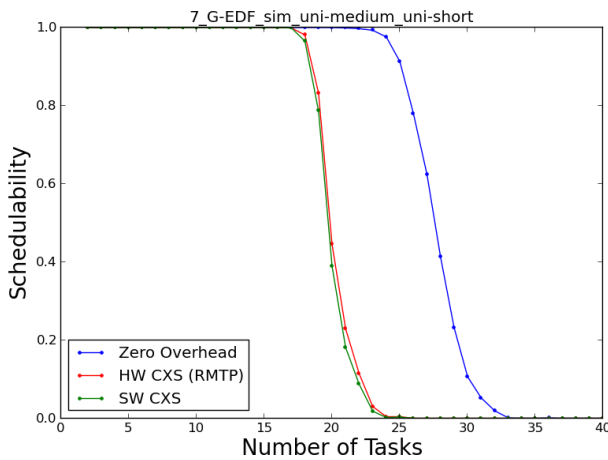


Figure 16: Schedulability Simulation on G-EDF (7 contexts)

5 Conclusions

In this research, we conducted an overhead-aware schedulability analysis on an 8-way SMT processor, D-RMTP. The evaluation shows that the runtime overheads have a crucial effect on systems in terms of schedulability. In order to predict the behaviour of the system, it can be said that considering runtime overheads is important. By considering the overheads, it is possible that the given task set cannot be run even if it is schedulable theoretically. In addition, by using the unique functionality of D-RMTP, the context cache, we showed that it is possible to improve schedulability.

For future work, we plan to evaluate the schedulability with more than 40 tasks. Since the size of context cache is 40, systems with more than 40 tasks have to save and restore the threads through memory. Since the overhead of context switch through memory is considerably high, the use of the context cache is important. Therefore, an efficient algorithm of utilizing the context cache is required. In addition, we plan to consider cache related overhead in the schedulability analysis because it is not negligible as well as the other overheads considered in this research.

Acknowledgment

This research was supported by CREST, JST.

References

[1] S.K. Baruah, J.E. Gehrke, and C.G. Plaxton. Fast Scheduling of Periodic Tasks on Multiple Resources. In *Parallel Processing Symposium*,

1995. *Proceedings., 9th International*, pages 280–288, Apr 1995.

[2] B.B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[3] B.B. Brandenburg and J.H. Anderson. On the Implementation of Global Real-Time Schedulers. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 214–224, Dec 2009.

[4] B.B. Brandenburg, H. Leontyev, and J.H. Anderson. An Overview of Interrupt Accounting Techniques for Multiprocessor Real-Time Systems. *Journal of Systems Architecture*, 57(6):638–654, 2011.

[5] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUSRT: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, December 2006.

[6] G. Giovani, F.A. Augusto, P. Rodolfo, and F. Sebastian. Implementation and Evaluation of Global and Partitioned Scheduling in a Real-Time OS. *Real-Time Systems*, 49(6):669–714, 2013.

[7] B. Marko and S. Baruah. Tests for Global EDF Schedulability Analysis. *J. Syst. Archit.*, 57(5):487–497, May 2011.

[8] K. Suito, R. Ueda, K. Fujii, T. Kogo, H. Matsutani, and N. Yamasaki. The Dependable Responsive Multithreaded Processor for Distributed Real-Time Systems. *Micro, IEEE*, 32(6):52–61, Nov 2012. RMTP.

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

[10] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-case Execution-time Problem; Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–53, May 2008.

[11] N. Yamasaki. Responsive Multithreaded Processor for Distributed Real-Time Systems. *Journal of Robotics and Mechatronics*, 17(2):130–141, 2005.