RTSS 2009
30th IEEE Real-Time Systems Symposium

Work-In-Progress Proceedings

December 1-4, 2009
Washington, D.C., USA

Sponsored by the IEEE Computer Society
Technical Committee on Real-Time Systems

Edited by Dakai Zhu

RTSS'09 Organizers

General Chair

Steve Goddard, University of Nebraska-Lincoln, USA

Program Chair

Theodore P. Baker, Florida State University, USA

Local Arrangements Chair

Hakan Aydin, George Mason University, USA

Special Track Chairs

Marilyn Wolf, Georgia Institute of Technology, USA (Cyber-Physical Systems)
Oleg Sokolsky, University of Pennsylvania, USA (Design and Verification of Embedded Real-Time Systems)
Chenyang Lu, Washington University, St. Louis, USA (Wireless Sensor Networks)

Work-in-Progress Chair

Dakai Zhu, University of Texas at San Antonio

Ex-Offico (TC Chair)

Raj Rajkumar, Carnegie Mellon University, USA

Finance Chair

Christopher Gill, Washington University, USA

Program Committee for Work-in-Progress

Hakan Aydin	George Mason University, USA
Enrico Bini	Scuola Superiore Sant'Anna, Italy
Jian-Jia Chen	Swiss Federal Institute of Technology, Switzerland
Qingxu Deng	Northeastern University, China
Nathan W. Fisher	Wayne State University, USA
Aniruddha Gokhale	Vanderbilt University, USA
Sathish Gopalakrishnan	University of British Columbia, Canada
Zonghua Gu	Zhejiang University, China
Shinpei KATO	University of Tokyo, Japan
Ying Lu	University of Nebraska - Lincoln, USA
Christian Poellabauer	University of Notre Dame, USA
Harini Ramaprasad	Southern Illinois University, USA
Shangping Ren	Illinois Institute of Technology, USA
Ali Tosun	University of Texas at San Antonio, USA
Wei Zhang	Southern Illinois University, USA

Table of Contents

Prioritized Out-of-Order Instruction Dispatching Techniques for Simultaneous Multi-Threading (SMT) Processors

Monobrata Debnath Byeong Kil Lee Wei-Ming Lin

Department of Electrical and Computer Engineering,
The University of Texas at San Antonio
{Monobrata.Debnath, Byeong.Lee, WeiMing.Lin}@utsa.edu

Abstract

Simultaneous multithreading provides an improved technique to increase resource utilization capability by sharing key datapath components among multiple independent threads. This thread-level parallelism (TLP) can be further exploited in the round robin dispatching with operand availability checking. The selection criteria for allocating Issue Queue (IQ) slots can be improved by not limiting the operand availability within the same thread. In this paper, we propose an effective scheduler for the SMT, named as Round robin with Operand Check that allocates IQ entries based on a round robin principle within a cycle. This scheme will dispatch at most one instruction from each thread at its turn, instead of dispatching all the available instructions from a single thread. Our approach shows a 12% performance (IPC) improvement with a smaller IQ size (16). The proposed scheme is also much better matched with small-scale processors that require a smaller IQ size (e.g., mobile processors).

1. Introduction

Simultaneous multithreading (SMT) offers an improved technique to take the traditional superscalar processors one step forward. The most common characteristic of SMT processors is the sharing of key datapath components among multiple independent threads, which ensures improved resource utilization. SMT not only exploits thread-level parallelism (TLP) among the various threads [1][2], but also equally concentrates on the advantages available at the instruction level in each thread in terms of parallelism (ILP). Consequently, the amount of hardware required of an SMT system is significantly less than a traditional superscalar machine to generate the same performance. Thus SMT passes the litmus test of conventional tradeoff between speed and cost. Most common resources which are shared in SMT technique are Issue Queue (IQ), physical register bank, various functional units and cache memories. Due to the requirement in resource sharing, these hardware components tend to remain busy in order to allocate large number of entries. These entries ensure the full performance potential afforded by SMT [4]. But these large numbers of entries among the shared resources exaggerate both the critical timing path and processor's cycle negatively. To retain the exploitation of thread-level parallelism (TLP) as well as instruction-level parallelism (ILP), a necessary solution must be introduced in order to minimize the complexity among these shared resources without affecting the ILP exploitation significantly.

One of the critical datapath structures in a Simultaneous Multithreading environment that might become a potential performance bottleneck is the IQ, and thus it requires an intelligent dispatching algorithm to alleviate this bottleneck. Most of the solutions to address this issue actually concentrate mostly on the quality of instructions fetched from different threads into the pipeline [3]. These solutions completely ignore how many operands are ready in an instruction. This check for readiness has the potential to reduce the IQ clogging because of potentially long latency instructions waiting to receive its first ready-operand, a cause for pipelining bottleneck. Another solution proposed by Sharkey et. al., 2_OP block [4], uses the readiness parameter and dispatches an instruction only when it has at least *one ready-operand.* Also it blocks a thread from any future dispatching soon after scheduler encounters an instruction with two non-ready-operands ignoring the instructions with one or two ready-operands which may appear next to the instruction with two non-ready-operands. This causes lower throughput at smaller IQ size and performance saturation with lower IPC than baseline SMT. We are focused on improving performance at smaller IQ size and moving saturation point with higher IPC.

Similar approaches in [3][5] optimize instruction scheduling with the combination of in-order and out-of-order execution. Studies also show that the delay of the select and wake-up logic are logarithmic and quadratic in nature with respect to instruction scheduling window size [6]. Moreover, power consumption of a large size instruction scheduling window cannot be ignored [7][8].

In this paper, we propose an effective scheduler for the SMT, named as *Round robin with Operand Check (RROC)*, which allocates the IQ entry based on a round robin principle. This scheme will dispatch at most one instruction from each thread at its turn, instead of dispatching all the available instructions from a single thread. Our approach shows a 12% performance (IPC) improvement with a smaller IQ size (16). This scheme

will be well fit for small-scale processors that require a smaller IQ size (e.g., mobile processors). Simulation results also show the performance improvement with variations of issue width and ROB size.

The rest of paper is organized as follows. Section 2 describes our proposed techniques. The workloads and simulation methodologies are introduced in section 3. Section 4 shows performance and power analysis, and finally, concluding remarks and future work are presented in the last section.

2. Prioritized Out-of-Order Instruction Dispatching Techniques

For instruction dispatching, we exploit dynamic information available immediately after the register renaming stage. Instructions with non-ready-operands usually spend a large number of clock cycles in the IQ in order to receive its operands [12]. This paper mainly concentrates on when and how the instructions should be placed in the IQ using micro-architectural information in runtime. The proposed *RROC* scheme successfully reduces the pressure on IQ by prioritizing the occupancy order of the IQ slots by instructions among all the threads instead of instructions from one thread. The key aspects of this paper are as follows:

- The proposed scheduler *RROC* designed for the SMT processor allocates instructions into the IQ based on round robin principle [9][10][11]. Instead of dispatching all the available instructions from a single thread within a dispatch cycle, all threads will take turn dispatching their ready instructions one at a time at their respective turn.
- Within each thread, a dispatching order is maintained among all ready-to-dispatch instructions based upon the number of ready-operands of each instruction. An instruction with two ready-operands receives the highest priority (assuming an ISA which supports two source operand instructions) and obtains the IQ entry earlier than instructions with one or zero ready-operand. Similarly, an instruction with only one ready-operand gets dispatched before the instructions with no ready operands.
- The scheduler blocks the thread which has zero instruction ready to dispatch in the current clock cycle. If all the threads are blocked due to unavailable ready-to-dispatch instructions, scheduler then jumps to the next clock cycle without dispatching any more instruction in the current clock cycle.

Figure 1 shows a 4 threaded SMT instruction scheduler using the proposed *RROC* scheme. Each thread is attained in round robin fashion to dispatch its native instructions. Once a thread is selected, the "Operand Check and Priority Set" module scans for available ready to dispatch instructions and their corresponding number of ready operands. Then a priority is set among all the instructions, based upon the number of ready-operands an instruction has. The new prioritized order decides which instruction should occupy the IQ entry first. Instructions with 2 ready-operands receive a highest priority followed by instructions with one and then zero ready-operands. Instructions with same number of ready-operands are chosen according to the order they appear.

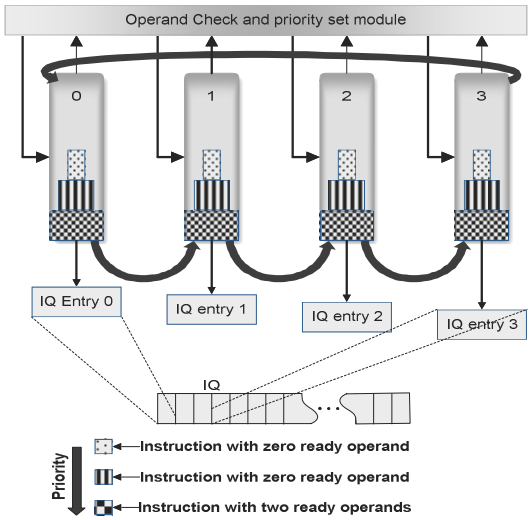


Figure 1. 4-threaded SMT with round robin and operand check scheduling technique

3. Workloads and Simulation Methodology

3.1 Workloads

For multi-threaded workloads, we use the mixed SPEC CPU 2000 benchmark suite [17] based on ILP classification. Each of the benchmarks is initialized in accordance with the procedure mentioned in Simpoints tool [16] and then 100 million instructions are simulated. Once the first 100 million instructions are committed from any of the threads, simulation is terminated.

Table 1. Simulated multi-threaded workloads

Classification	Mix	Benchmarks
2 Low ILP + 2 High ILP	Mix1	crafty,swim,mesa, perlbmk
	Mix4	mcf,equake,mesa, crafty
4 Low ILP	Mix2	mcf,equake,vpr,lucas
4 Med ILP	Mix3	applu,ammp,mgrid, galgel
1 High ILP + 1 Low ILP + 2 Med ILP	Mix5	perlbmk,lucas,galgel, gcc
	Mix6	mesa, swim ,apsi, mgrid

In order to categorize multithreaded workloads, each of the benchmarks is simulated individually in the Simple scalar [15] environment. As shown in Table 1, three types of ILPs – low ILP (memory bound), medium ILP and high ILP (execution bound) – are identified, and 6 mixed multi-threaded workloads are created based on different combinations of ILP types.

3.2 Simulation Environments

We used the tool M-Sim[14], a multi-threaded micro architectural simulation environment model, to estimate performance and power analysis of the proposed scheme. M-sim includes accurate models of the pipeline structures such as explicit register renaming, concurrent execution of multiple threads, detailed power estimation using Wattch framework [18], separate Reorder Buffer (ROB), and register files which are necessary for Simultaneous Multithreading (SMT) model. The Issue Queue, execution units, Load-Store Queue (LSQ) are shared among the threads, but branch predictor is exclusive to each thread. The detailed processor's configuration is shown in Table 2.

Table 2. Configuration of the simulated processor

Parameter	Configuration
Machine Width	8 wide fetch / issue / commit
Window size	Issue queue (16-entry for best performance), 48 entry Load/Store queue, 128-entry ROB or larger
Function Units and Latency (total/issue)	8 Int Add (1/1), 4 Int Mult (3/1) / Div (20/19), 4 Load/Store (2/1), 8 FP Add (2), 4 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
Physical registers	256 integer and floating point
L1 I–cache	64KB, 2–way set–associative, 128 byte line
L1 D–cache	32 KB, 4–way set–associative, 256 byte line
L2 Cache unified	2 MB, 8–way set–associative, 512 byte line
BTB	2048 entry, 2–way set–associative
Branch Predictor	2K entry gShare, 10-bit global history per thread
Pipeline Structure	5-stage front-end (fetch-dispatch), scheduling (for register file access -2 stages, execution, write back, commit)
Memory	64 bit wide, 200 cycles access latency

4. Result Analysis

4.1 Performance Evaluation

Our proposed scheme shows a significant improvement in terms of average throughput (IPC). In our simulation, we perform three different analyses in comparing IPC by varying ROB size, IQ entry size and instruction issue-width, respectively. Figure 2 and 3 compare the performance impact of the proposed scheme with different issue width and ROB size in terms of average throughput, with the other two hardware parameters unchanged. The proposed dispatching technique shows a performance improvement from 10% to 25% with different issue-widths, and an improvement from 10% to 16% when varying the size of ROB. Table 3 and figure 4 reflect the impact of the proposed scheme under different entry size IQ, with a fixed issue width (8) and a 96-entry ROB size. It also gives a performance comparison between the 2_OP block and the proposed *RROC*. We see that the proposed scheme delivers a performance improvement of around 12% compared to the traditional scheme with a small IQ size (16). It is interesting to note that the performance improvement is saturated as the size difference between the IQ and ROB decreases. How this size variation between the ROB and IQ exactly affects performance and power consumption is also being studied.

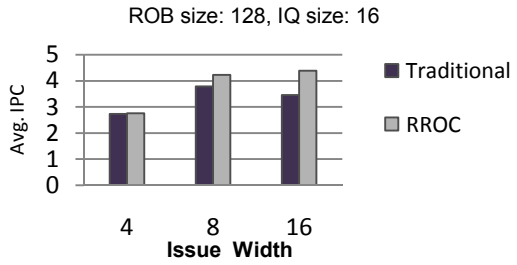


Figure 2. Average Throughput (IPC) vs. Issue width

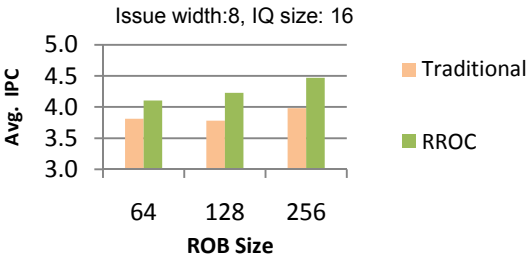


Figure 3. Change of throughput IPC for different ROB size

4.2 Power Analysis

It is important to note that how our proposed scheme affects power consumption. In this simulation, we compare average total power per cycle with the baseline machine. Figure 5 shows the comparison for different instruction widths. For all three types of simulations, power remained almost constant.

Table 3. Performance analysis with different IQ entry sizes

Bench mark Mix	IQ 16			IQ 32			IQ 48			IQ 64		
	Trd[1]	2op[2]	Roc[3]	Trd[1]	2op[2]	Roc[3]	Trd[1]	2op[2]	Roc[3]	Trd[1]	2op[2]	Roc[3]
1	4.25	4.57	4.67	4.83	4.68	4.81	4.84	4.69	4.81	4.84	4.68	4.81
2	3.13	2.98	2.98	2.98	2.97	2.97	2.96	2.97	2.97	2.98	2.97	2.96
3	4.49	4.78	5.28	5.49	5.38	5.49	5.54	5.39	5.50	5.55	5.39	5.51
4	2.74	2.73	2.75	2.75	2.74	2.75	2.76	2.74	2.75	2.76	2.74	2.75
5	4.24	4.28	4.39	4.59	4.40	4.54	4.60	4.40	4.55	4.60	4.41	4.56
6	4.17	4.82	5.17	5.31	5.11	5.34	5.33	5.11	5.34	5.34	5.12	5.35
Avg.	3.84	4.03	4.21	4.32	4.21	4.32	4.34	4.22	4.32	4.34	4.22	4.32

[1]: Traditional Scheduling , [2]: 2 _OP block scheduling, [3]: Round robin with operand checking scheduling.

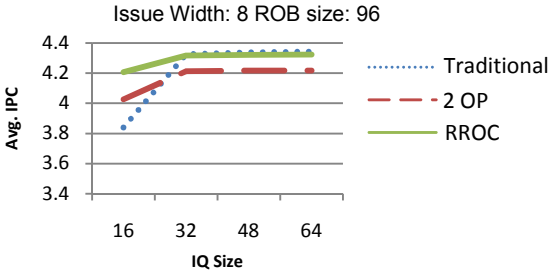


Figure 4. IPC variation for different IQ size

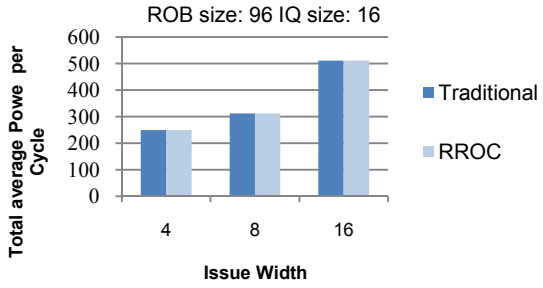


Figure 5. Power variation with issue-width

5. Conclusion and Future Work

The proposed dispatching algorithm in this paper has demonstrated its potential in leading to substantial performance improvement without compromising the power consumption issue. It allows for a higher degree of fairness among threads in instruction dispatching and better resource utilization at smaller IQ size. This new foundation will facilitate even more design improvement to be incorporated into the system. Potential future research directions include the study in determining how the performance is affected by the size variation between the ROB and IQ needs using the proposed scheduler. Also, circuit delays of the proposed scheme also need to be investigated with various configurations.

References

[1] D. Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," Proc. Int'l Symp. Computer Architecture, 1996.

[2] D. Tullsen et al., "Simultaneous Multithreading: Maximizing On-Chip Parallelism," Proc. Int'l Symp. Computer Architecture, 1995.

[3] J. Sharkey and D. Ponomarev, "Exploiting Operand Availability for Efficient Simultaneous Multithreading," IEEE Transactions on Computers, vol. 56, no. 2, February 2007.

[4] J. Sharkey and D. Ponomarev, "Efficient Instruction Schedulers for SMT Processors," Proc. 12th Int'l Symp. High Performance Computer Architecture (HPCA), 2006.

[5] Hui Wang, Rama Sangireddy, "Optimizing Instruction Scheduling through Combined In-Order and O-O-O Execution in SMT Processors," IEEE Transactions On Parallel And Distributed Systems vol. 20, no. 3, pp. 389-403, March 2009.

[6] Subbarao Palacharla. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," Proc. 24th Ann. Int'l Symp. Computer Architecture (ISCA '97), pp. 206-218, 1997

[7] N. Mehta, B. Singer, R.I. Bahar, M. Leuchtenburg, and R. Weiss,"Fetch Halting on Critical Load Misses," Proc. 22nd IEEE Int'l Conf.Computer Design (ICCAD), 2004

[8] D. Folegnani and A. Gonzalez, "Energy-Effective Issue Logic,"Proc. Int'l Symp. Computer Architecture, July 2001

[9] J. Laudon, A. Gupta, and M. Horowitz. "Interleaving: A multithreading technique targeting multiprocessors and workstations," 6th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1994.

[10] B. Boothe and A. Ranade. "Improved multithreading techniques for hiding communication latency in multiprocessors,"19th Annual International Symposium on Computer Architecture, 1992

[11] J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, J. Levy, and S. Parekh. "An analysis of database workload performance on simultaneous multithreading processors", 25nd Annual International Symposium on Computer Architecture, June 1998

[12] Yu-Lai Zhao, Xian-Feng Li, Dong Tong Xu Cheng ."An Energy-Efficient Instruction Scheduler Design with Two-Level Shelving and Adaptive Banking," Journal of Computer Science and Technology, Volume 22 Issue 1, pp. 206-218, January 2007.

[13] J. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium", in the Transactions of IEEE Computer, 33(7):2835, July 2000.

[14] J. Sharkey. "M-Sim: A Flexible, Multi-threaded Simulation Environment." Tech. Report CS-TR-05-DP1, Department of Computer Science, SUNY Binghamton, 2005.

[15] D. Burger, T. Austin. "The SimpleScalar tool set: Version 2.0." Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997 and documentation for all Simplescalar releases.

[16] T. Sherwood, et al. "Automatically Characterizing Large Scale Program Behavior," Proc. ASPLOS, 2002.

[17] Standard Performance Evaluation Corporation (SPEC) website, http://www.spec.org/

[18] D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," International Symposium on Computer Architecture (ISCA-27), June 2000

Finding an Upper Bound on the Increase in Execution Time Due to Contention on the Memory Bus in COTS-Based Multicore Systems

Björn Andersson and Arvind Easwaran
Polytechnic Institute of Porto, Portugal
bandersson@dei.isep.ipp.pt, aen@isep.ipp.pt

Jinkyu Lee
Dept. of Computer Science, KAIST, South Korea
jinkyu@cps.kaist.ac.kr

***Abstract*—Contention on the memory bus in COTS based multicore systems is becoming a major determining factor of the execution time of a task. Analyzing this extra execution time is non-trivial because (i) bus arbitration protocols in such systems are often undocumented and (ii) the times when the memory bus is requested to be used are not explicitly controlled by the operating system scheduler; they are instead a result of cache misses. We present a method for finding an upper bound on the extra execution time of a task due to contention on the memory bus in COTS based multicore systems. This method makes no assumptions on the bus arbitration protocol (other than assuming that it is work-conserving).**

I. Introduction

The multicore processor is today a generic building block in the design of embedded real-time computing systems. Typically, a multicore processor chip is comprised of a set of processor cores, each with a private cache memory (L1) and potentially a cache memory (L2) that is shared among the processor cores. This chip is connected through an interconnection network (such as a bus) to a set of main-memory modules. When an instruction cannot be served by the on-chip caches, it is necessary for the processor to perform a transaction on the interconnection network in order to fetch data from the main-memory modules. Already today, this interconnection network is a performance bottleneck for many applications [1]. Moreover, since the number of processor cores in a multicore chip is increasing dramatically, the amount of traffic on the interconnection network increases accordingly, and consequently this problem is expected to be exacerbated in the future [1], [2].

The interconnection network has an impact also on the execution time of an individual task. Consider a task τ_1 executing on processor P_1 and another task τ_2 executing on processor P_2. The task τ_2 generates a cache miss but before the transaction on the interconnection network has finished, task τ_1 generates a cache miss as well. Then in this case, serving the cache miss of τ_1 requires more time than would have been the case if τ_1 was the only task in the system because the bus must finish serving the other processor. Therefore, it is important to develop a method for finding an upper bound on the extra execution time of a task due to contention on the interconnection network between processor cores. This problem differs from studies in Worst-Case Execution-Time (WCET) analysis because WCET analysis is performed on a task in isolation, whereas our problem concerns the interaction of tasks. It is also different from works in real-time communication because these studies find an upper bound on the queuing time of individual message transmission requests, whereas our problem concerns the cumulative waiting time of many requests to perform transactions on the memory bus.

The scientific community has nonetheless provided some initial insights into the problem of contention on the interconnection network. The software of a task can be structured to be separated into a fetch phase (where cache misses are allowed) and an execution phase (where cache misses are not allowed) and then the memory bus and tasks are scheduled to ensure that no two processor cores are in a fetch phase simultaneously [3], thus eliminating contention. Or, a rate-limiter is added to the memory controller to ensure that no processor core will generate too much traffic in a time interval of pre-specified duration and then network calculus is used to analyze the processor cores [4]. Unfortunately, common to these approaches is that they require control of the arbitration of the memory bus and hence they cannot be used on COTS-based multicores.

In this work we develop the first approach for finding an upper bound on the extra execution time of a task due to contention on the memory bus in COTS-based multicore systems. We also present a technique for characterizing the interconnection-network-traffic generation pattern of tasks, which is then used as input for finding the upper bound on task execution time. In taking this first step, we make the following assumptions:

- A1. The interconnection network is a shared bus (denoted as *memory bus* henceforth);
- A2. The shared L2 cache is either partitioned between the processor cores or disabled if it cannot be partitioned. The rationale for this assumption is explained in Section II-C;
- A3. Tasks are statically assigned to processors and all jobs execute on the processor to which the task is assigned (partitioned scheduling);
- A4. Non-preemptive scheduling is used on each pro-

cessor;

A5. The bus arbitration protocol is work-conserving (that is, the memory bus is idle only if no processor core requests to use the bus).

Additionally, the technique developed in this paper has the following properties.

P1. It does not assume any specific arbitration protocol;

P2. It works for constrained-deadline sporadic tasks.

II. System Model

A. Task model

We assume that the workload is comprised of a set of tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. We assume the constrained-deadline sporadic task model which characterizes a task τ_i by C_i, T_i and D_i ($D_i \leq T_i$), with the interpretation that τ_i releases a sequence of jobs such that two subsequent jobs from τ_i are released at least T_i time units apart and the exact times of the releases of these jobs cannot be controlled by the scheduling algorithm. Each job released by τ_i requests to perform C_i units of execution at most D_i time units from its release; otherwise it misses its deadline.

Note that C_i denotes an upper bound on the execution time of a job of task τ_i when the job executes with no contention on the memory bus from other tasks. C_i can be found by WCET analysis techniques. In this work, we are interested in finding C'_i which denotes an upper bound on the execution time when the job executes *with* contention on the memory bus from other tasks on other processors.

B. Architecture

Many high-performance processors today allow more than one instruction to be issued in parallel, executed in parallel and committed in parallel. Our model allows this. Some high-performance processors allow instructions to be committed in another order than they were issued. Such processors can significantly complicate WCET analysis [5] even on a single processor without bus contention. Therefore, we assume that processor cores are in-order processors. Also, some processors may switch to another thread when a long-latency instruction is executed (for example a data-cache miss). We assume this is not the case, that is, we assume that when a processor core waits for accessing the memory bus, the processor core is simply stalled. For these reasons, we can compute C'_i as $C'_i = C_i + Q_i$, where Q_i is an upper bound on the amount of time that task τ_i stalls execution when it executes for C'_i time units because of waiting for accessing the memory bus.

Some computer systems use split transaction buses, where a memory transaction is split into a request part (address) and a reply part (data). We assume that the computer system does not use split transactions.

Different bus transactions may take different amount of time. For example, a bus transaction resulting from a load which succeeds another load instruction with the same row-address can be served faster because only the column address of the DRAM memory needs to be changed. We let TR denote an upper bound on the amount of time for performing a bus transaction. This time, TR, includes the time to assert the address, the time for the memory latency of the main-memory module and the time for the main-memory module to deliver the data to the processor. Note that TR denotes an upper bound on the amount of time for performing a bus transaction for the case that the bus was idle; therefore, TR does not include any queuing delay on the memory bus.

We make no assumption on the number of processor chips or the number of processor cores per processor chip. For example, we allow systems with a single processor chip comprising two processor cores. We also allow systems with two processor chips, each comprising four processor cores. Further, we assume that tasks are already assigned to processor cores. Therefore, we let τ^p denote the set of tasks assigned to processor core P_p.

Because of our definition of C'_i it holds that if a task executed on a processor for C'_i time units, then it performs C_i units of execution. Therefore, we can check schedulability of all tasks assigned to processor p by using a uniprocessor schedulability test on processor p, but for each task τ_i, replace C_i by C'_i. See for example [6].

C. Bus requests

We assume that $BR_i(t)$ is a function that denotes an upper bound on the number of bus requests that task τ_i can generate during any time interval of duration t. We will use $BR_i(t)$ in Section III for computing C'_i. It is also necessary to find $BR_i(t)$; Section IV shows this.

The function $BR_i(t)$ is clearly dependent on task τ_i. The task generates a bus request when it misses the shared L2 cache and before that it must also have missed its private L1 cache. Note that since the L2 cache is shared, the function $BR_i(t)$ is actually not only a property of task τ_i and the caches but it is also a property of the interaction between task τ_i and the tasks on other processor cores. This makes the analysis very complicated. Therefore, in order to simplify our study, we made assumption A2 regarding the shared L2 cache (partitioned or disabled). As a result $BR_i(t)$ does not depend on the behavior of tasks in other processor cores.

D. Problem statement

Our problem can therefore be stated as:

> Given $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where each τ_i is characterized by T_i, D_i and C_i and the function $BR_i(t)$, and given that each task is assigned to a processor and executes on it non-preemptively, find C'_i for each τ_i.

III. Analysis on the Worst-Case Execution Time with Contention on the Memory Bus

In this section, we show how to calculate an upper bound on the execution time when the job executes with contention on the memory bus from other tasks on other processors (*i.e.*, C_i') using given $BR_i(t)$. We only assume that the bus arbitration protocol is work-conserving, and do not assume any other specific arbitration protocol.

Consider a job $J_{i,k}$ released by τ_i. Bus transactions requested from $J_{i,k}$ during its execution can be delayed due to: (a) bus transactions requested from other jobs on other processors during the execution of $J_{i,k}$ and (b) any bus transactions requested but not performed before the execution of $J_{i,k}$ (backlogged bus transactions). Considering this delay, we can calculate C_i' as follows.

$$C_i' = C_i + BL_i \cdot TR + \sum_{\tau_j \in (\tau \setminus \tau^{proc(\tau_i)})} BR_j(C_i') \cdot TR \quad (1)$$

where BL_i is the maximum number of backlogged bus transactions at the beginning of the execution of a job released by τ_i, and $proc(\tau_i)$ is a processor to which τ_i is assigned. We consider (a) by $\sum_{\tau_j \in (\tau \setminus \tau^{proc(\tau_i)})} BR_j(C_i') \cdot TR$. Here note that we use $BR_j(C_i')$ instead of $BR_j(C_i)$, so that we can care for additional bus requests caused by additional execution time ($C_i' - C_i$). We also consider (b) by $BL_i \cdot TR$, but we need to know how large BL_i is.

To calculate BL_i, we first define the maximum busy period of bus transactions (denoted as BP), and BP can be calculated by the following recurrence equation.

$$BP = TR + \sum_{\tau_j \in \tau} BR_j(BP) \cdot TR \quad (2)$$

The structure of this equation is similar to that of calculating response time. We know $BR_j(\cdot)$ is a non-decreasing function so that we calculate Eq. (2) in an iterative manner (*i.e*, $BP^{(k+1)} = f(BP^{(k)})$ starting from $BP^{(0)} = TR$. If we replace BP with t in Eq. (2), the right-hand side of the equation means the longest time interval of bus transactions requested during t. If we consider that bus transactions are continuously performed during the busy period, BL_i can be calculated by the following equation.

$$BL_i = \max_{0 \le t \le BP} \left\lceil \frac{TR + (\sum_{\tau_j \in \tau} BR_j(t) \cdot TR) - t}{TR} \right\rceil \quad (3)$$

Now we know all variables in Eq. (1) and thus calculate the equation similar to Eq. (2) since the right-hand side of the equation is a non-decreasing function of C_i'.

IV. Analysis on the Maximum Number of Bus Requests

In this section, we show how to calculate a function which is an upper bound on the number of bus requests that task τ_i can generate during any time interval of duration t (*i.e.*, $BR_i(t)$) from experimental results.

We assume the following can be obtained from experiments:

1) An upper bound on the number of bus requests in an interval $[0, t]$ (denoted as $ARH_i^j(t)$), where 0 denotes the beginning of execution of the j^{th} execution path of τ_i and t denotes some future time. Note that this measurement only depends on the execution of task τ_i, because of our assumption A2 on the shared L2 cache. Therefore, we obtain $ARH_i^j(t)$ from measurements by executing jobs of τ_i independently on a processor core;
2) An lower bound on the number of bus requests in an interval $[0, t]$ (denoted as $ARL_i^j(t)$), where 0 denotes the beginning of execution of the j^{th} execution path of τ_i and t denotes some future time;
3) The execution time of the j^{th} path of task τ_i (denoted as C_i^j).

We note that different executions of the same path may result in different numbers of bus requests; this is the reason why we distinguish between $ARH_i^j(t)$ and $ARL_i^j(t)$. We let $paths(\tau_i)$ denote the set of all paths of task τ_i. We assume that $ARH_i^j(t)$ and $ARL_i^j(t)$ are non-decreasing functions for each i and j and regard C_i as $\max_{j \in paths(\tau_i)} C_i^j$.

To calculate $BR_i(t)$, we divide t into three parts: the head, middle, and tail parts, and denote them as $f_i^H(t_H)$, $f_i^M(t_M)$, and $f_i^T(t_T)$, respectively. This idea of splitting a path to simplify the analysis has been used before; for instance, in [7], for analysis of recurring task models. As shown in Figure 1(a), the duration of the middle part is a multiple of T_i so that there exist several complete executions. The duration of the head is less than T_i, so that there exists either one partial execution or one complete execution. Ditto for the tail. The head (tail) part includes the end (beginning) point of an execution as shown in Figure 1(a). Since the definition of $BR_i(t)$ is an upper bound on the number of bus requests, it can be calculated as follows.

$$BR_i(t) = \max_{t_H, t_M, t_T} f_i^H(t_H) + f_i^M(t_M) + f_i^T(t_T), \quad (4)$$

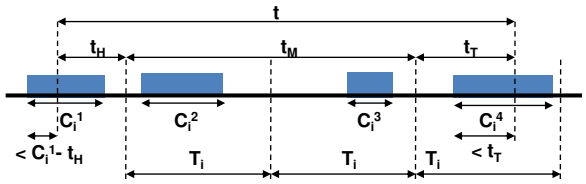
$$\text{where } t_H + t_M + t_T = t, \quad (5)$$

$$t_H, t_T < T_i, \quad (6)$$

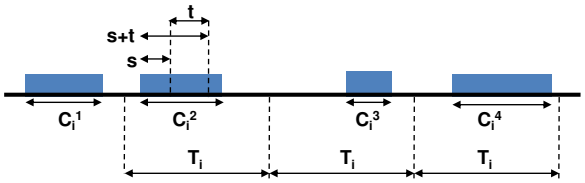
$$t_M = \left[\left\lfloor \frac{t}{T_i} \right\rfloor - 1 \right]^+ T_i \text{ or } t_M = \left\lfloor \frac{t}{T_i} \right\rfloor T_i \quad (7)$$

where $[A]^+$ means $\max\{A, 0\}$. Note that Eq. (7) is derived from Eq. (5) and (6). Also note that t_M is a multiple of T_i.

The head part starts from any arbitrary point of an execution but includes the end point of the execution, so that the maximum number of bus requests of the head part is equal to the maximum difference between the number of bus requests during a complete execution and the one during a partial execution.



(a) For the general case



(b) For $t < T_i$

Figure 1. Calculation of $BR_i(t)$

$$f_i^H(t_H) = \max_{j \in paths(\tau_i)} ARH_i^j(C_i^j) - ARL_i^j([C_i^j - t_H]^+) \quad (8)$$

In the middle part, there exists exactly one instance of execution for every T_i, and thus we choose the maximum execution time among all paths.

$$f_i^M(t_M) = \frac{t_M}{T_i} \cdot \max_{j \in paths(\tau_i)} \{ARH_i^j(C_i^j)\} \quad (9)$$

The tail part includes the beginning point of an execution but ends at any arbitrary point, so we can calculate the maximum number of bus requests of the tail part similarly to that of the head part.

$$f_i^T(t_T) = \max_{j \in paths(\tau_i)} ARH_i^j(\min\{t_T, C_i^j\}) \quad (10)$$

Eq. (4) assumes that the head (tail) part includes the end (beginning) of an execution. If t is smaller than T_i, there might be less than one entire execution during t. In this case, the assumption of Eq. (4) is broken, so we need to analyze $BR_i(t)$ in a different way. Once we consider the time interval that starts and ends at an arbitrary point of an execution as shown in Figure 1(b), $BR_i(t)$ can be found as:

$$BR_i(t) = \max_{j \in paths(\tau_i), 0 \le s \le C_i} ARH_i^j(\min\{s+t, C_i^j\}) - ARL_i^j(s) \quad (11)$$

Finally, we can calculate $BR_i(t)$ in the following ways: if $t \ge T_i$, use the result of Eq. (4); and if $t < T_i$, choose the maximum value between Eq. (4) and Eq. (11).

V. Conclusions and Future work

We have presented an approach for finding an upper bound on the extra execution of a task due to contention on the interconnection network between processor cores and memory in a COTS-based multicore system. To the best of our knowledge, the problem of analyzing such extra execution time was previously unsolved. Now, we are taking measurements of real programs and using them to build a model of the bus request pattern for each program. When we finish it, we take measurements of the response time of programs to test if our proposed methods for calculating an upper bound on the extra execution time of a task is valid in practice.

This paper is a starting point of considering the effect of bus contention for real-time tasks on multicores, and thus there are many potential research issues. Our future work includes (i) allowing preemptive scheduling, (ii) analyzing switched interconnection networks, (iii) avoiding the pessimism resulting from each task being analyzed individually (that is, task τ_1 has to wait for all bus transactions from τ_2, and task τ_2 has to wait for all bus transactions from τ_1.), and (iv) developing processor scheduling algorithms that facilitate the analysis of the memory bus contention.

References

[1] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[2] K. D. Bosschere, W. Luk, X. Martorell, N. Navarro, M. OBoyle, D. Pnevmatikatos, A. Ramirez, P. Sainrat, A. Seznec, P. Stenström, and O. Temam, "Challenge 2.2 in high-performance embedded architecture and compilation roadmap," in *Transactions on HiPEAC*, 2007, pp. 5–29.

[3] J. Rosén, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *Proc. of IEEE Real-Time Systems Symposium*, 2007, pp. 49–60.

[4] L. Steffens, M. Agarwal, and P. van der Wolf, "Real-time analysis for memory access in media processing SoCs: A practical approach," in *Proc. of Euromicro Conference on Real-Time Systems*, 2008, pp. 255–265.

[5] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *Proc. of IEEE Real-Time Systems Symposium*, 1999, pp. 12–21.

[6] G. Laurent, N. Rivierre, and M. Spuri, "Preemptive and non-preemptive real-time uniprocessor scheduling," Tech. Rep., 1996.

[7] S. K. Baruah, "A general model for recurring real-time tasks," in *Proc. of IEEE Real-Time Systems Symposium*, 1998, pp. 114–122.

Acknowledgements

This work was partially supported by ARTISTDesign Network of Excellence on Embedded Systems Design, funded by the European Commission under FP7 with contract number ICT-NoE-214373 and the Portuguese Science and Technology Foundation (Fundação para Ciência e Tecnologia - FCT). This paper has been produced partially in a time when we have had general bus-contention discussions with Stefan Petters and Dakshina Dasari.

Time-Predictable and High-Performance Cache Architectures for Multicore Processors

Jun Yan, Wei Zhang and Yu Liu
Department of Electrical and Computer Engineering
Southern Illinois University, Carbondale, IL 62901
{jun,zhang,liu}@engr.siu.edu

Abstract

This paper studies several time-predictable cache architectures to guarantee time predictability for real-time threads without significantly impacting the performance (i.e., throughput). We propose a prioritized cache that gives priority to real-time threads while allowing all the threads to share the aggregate cache space. Also, we study a prioritized-partitioned cache to provide decent performance to non-real-time threads without compromising the time predictability of real-time threads. Our experiments indicate that the proposed time-predictable caches can be used for different real-time applications with various cache access behaviors for balancing time predictability and performance.

1. Introduction

In a multi-core processor, in addition to the complexity of WCET analysis for a single core, different cores typically share resources such as the low-level cache. Therefore, threads running on different cores may interfere with each other in accessing the shared resources at runtime, which is very difficult to be analyzed statically and accurately. To ensure reliable use of multi-core computing platforms for various real-time applications, this paper proposes a prioritized cache architecture to ensure time predictability of real-time threads while still allowing other (non-real-time) threads to share cache space for maximizing performance. Also, we explore partitioned caches and prioritized-partitioned caches to better balance performance and time predictability for various real-time applications. In addition, we have developed a WCET analysis tool based on Chronos [4] to reasonably bound the worst-case performance for multi-core processors with time-predictable caches.

2. Time-Predictable Caches for Multi-Core Chips

2.1 Separated Caches

Obviously, a straightforward solution to enhance time predictability of multi-core caches is to use separated caches, as shown in Figure 1 (a). Separated caches make WCET analysis simpler because inter-core (L2) cache interferences are completely avoided. On the other hand, separated caches have several drawbacks. First, the performance may be inferior (though predictable), especially when the working set of a real-time thread is larger than the size of its private L2 cache. Second, separated L2 caches may waste the precious cache space due to both internal cache fragmentation within each core and the possible redundancy across separated caches of different cores because of the reuse of instructions and/or data, which also makes it more expensive to maintain cache coherence. Also, separate caches are rigid and thus may not be attractive for emerging real-time applications with a mix of real-time threads and non-real-time threads.

2.2 Partitioned Caches

Cache partitioning is not a new idea. Recently cache partitioning [3, 5] has been studied intensively to enhance fairness or QoS for multi-core chips. Nesbit et al. [8] proposed virtual private caches, which can achieve QoS for the shared cache in multicore systems by improving fairness in bandwidth and capacity management. However, most of prior studies have focused on studying general-purpose applications, not specifically for real-time applications. Generally, cache partitioning can be implemented by using a global approach [5] or a per-set approach [3]. In the partitioned cache architecture, since each core has its own portion of the partitioned cache, the inter-core cache interferences are avoided, hence leading to high time predictability. At the same time, non-real-time threads can still have decent performance, because their own cache space cannot be invaded by real-time threads. Moreover, compared with separated caches, partitioned caches are more flexible, because the aggregate cache space can be partitioned differently based on the cache demands of various threads running on different cores. For instance, as shown in Figure 1 (b), suppose a real-time memory-intensive thread is running on core 2, it is possible to allocate more L2 space to core 2 than core 1. In addition, the cache space for non-real-time threads can

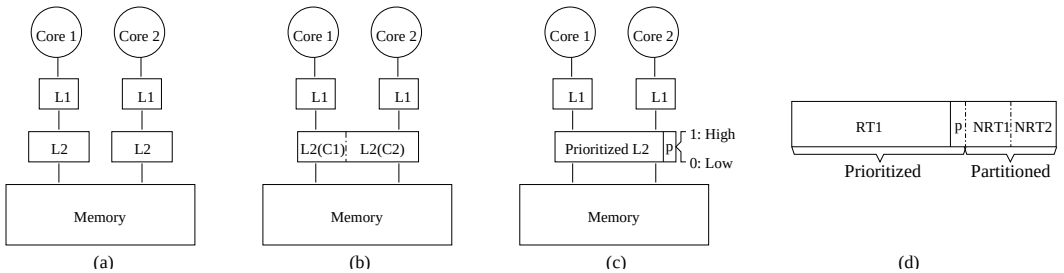


Figure 1. (a) separated L2 caches, (b) a partitioned L2 cache, (c) a prioritized L2 cache, and (d) a prioritized-partitioned cache. Note RT represents a real-time thread, and NRT represents a non-real-time thread.

be adjusted dynamically and adaptively to maximize performance.

2.3 Prioritized Caches

In this paper, we propose a prioritized cache, which is both time-predictable (for the high-priority thread) and sharable by different threads, including non-real-time threads. In a prioritized cache, all the threads can use the shared cache space; however, threads running on different cores are assigned different priorities by the O.S. Accordingly, a thread with lower priority cannot replace data/instructions of a high-priority thread, while a thread with higher priority can overlap data/instructions of a low-priority thread. Therefore, in a prioritized cache, the memory accesses of a high-priority thread will not be affected by the memory accesses of low-priority threads at runtime, thus making reasonable WCET analysis for the high-priority thread possible. At the same time, the low-priority threads can still use the shared cache space that is available, unless all the cache blocks have been used by the high-priority thread, which is very unlikely as the worst-case cache behavior rarely happens.

As can be seen from Figure 1 (c), a priority bit is added to each cache line of the prioritized L2 cache, which is used to differentiate the priorities of threads from different cores. Specifically, 1 can be assigned to the cache blocks that are used by high-priority threads, and 0 is associated to the cache blocks of low-priority threads. In mixed real-time applications the O.S (or programmers) can assign high priority (i.e., 1) to real-time threads and low priority (i.e., 0) to non-real-time threads. By default, at the time of each cache replacement, the priority bit of any data or instruction will be set based on the priority of the thread that accesses it. However, to enable data sharing between different threads, when there is a hit of a low-priority cache block by the high-priority thread, the high-priority thread can overlap the priority bit of the existing cache block, so that it will not be unfairly replaced due to its low-priority. In contrast, a low-priority thread cannot overwrite the priority bit of higher priority. Also, when there is a hit in the prioritized cache, the data and/or instructions can be accessed by any thread, regardless of their priorities. Accordingly, the cache replacement algorithm needs to be modified to ensure the priority by using the priority bit.

It should be noted that Tan and Mooney [7] proposed a prioritized cache for multi-tasking systems. In contrast, this paper studies prioritized caches for multicore processors, where different cores can access the shared cache simultaneously. Another difference is that in Tan's work [7], tasks with different priorities can only use pre-partitioned columns; while in our design, they can use any cache block mapped.

	size	bsize	assoc	latency
L1-i-cache	4K	32	2	10
L1-d-cache	perfect			
L2-u-cache	32K	32	8	100

Table 1. Configurations of the dual-core chip memory hierarchy.

Moreover, we have develop WCET analysis to bound the worst-case performance of the prioritized caches proposed in this paper.

2.4 Prioritized-Partitioned Caches

While the prioritized cache architecture can improve the time predictability of the high-priority thread, the threads with low priority may suffer from large performance loss if the high-priority thread is memory-intensive. To ensure reasonable performance for the non-real-time threads without impacting time predictability of real-time threads, we propose to incorporate cache partitioning into the prioritized cache architecture, which is also called prioritized-partitioned caches in this paper. As can be seen from Figure 1 (d), in a prioritized-partitioned cache, the aggregate cache space is first partitioned based on the demands of real-time and non-real-time threads, with preference given to real-time threads so as to guarantee their meeting of the deadline constraints. After the partition, the space reserved for non-real-time threads cannot be invaded by real-time threads any more. However, in contrast to pure partitioned caches, the unused or under-utilized cache space by real-time threads can still be exploited by non-real-time threads. In other words, non-real-time threads can virtually use all the cache space; however, they have lower priority in accessing the cache space allocated to real-time threads.

3. Evaluation Methodology

The WCET analysis for the time-predictable multi-core processors is based on an extended Chronos timing analysis tool [4] as aforementioned. We use SESC simulator to simulate a baseline dual-core processor, in which each core is a 4-issue superscalar processor with 5 pipeline stages, 32 integer registers and 32 floating-point registers. The important parameters of the dual-core memory hierarchy are given in Table 1. The benchmarks are selected from MediaBench (as real-time threads) and SPEC 2000 (as non-real-time benchmarks or real-time threads with a lower priority, including `164.gzip` and `255.vortex`).

In our evaluation, we consider the following four schemes:

- *Base:* In the base scheme, we assume only a single thread is running on the baseline dual-core with a shared L2 cache.
- *Prioritized:* This is the scheme that uses the prioritized cache.
- *Partitioned:* This is the scheme that uses the partitioned cache. By default, the total cache space is equally partitioned among different cores.
- *Prioritized-Partitioned (PP):* This is the scheme that uses the prioritized-partitioned cache. By default, the total cache space is equally partitioned among different cores. In this scheme, the first thread (i.e., the leftmost thread in the group) has the highest priority; the second thread has the second highest priority; and so on.

4. Experimental Results

4.1 Performance Results

In performance evaluation, we first study the performance of the prioritized cache. We select `vortex` as the non-real-time benchmark, which is run concurrently with another MediaBench that serves as the real-time application. The combination of a real-time thread and a non-real-time thread running on the baseline dual-core is also called a pair in this paper. Figure 2 demonstrates the performance of the prioritized cache for both the real-time thread (RTT) and `vortex`, which are normalized with their execution cycles in the base scheme. For all these eight pairs of applications, we find the high-priority thread can always achieve the same performance as the base scheme, indicating that the prioritized cache can successfully avoid interferences from the co-running non-real-time thread. On the other hand, we find the performance of the non-real-time thread, i.e., `vortex`, degrades dramatically in many pairs due to its low priority in accessing the prioritized cache. Specifically, the normalized performance degradation (i.e., difference between the prioritized cache and the base scheme) of `vortex` varies from 8.3% (for `cordic-vortex`) up to 280% (for `des-vortex`). Interestingly, we also observe that when `vortex` is run with a *low*-utility real-time application such as `cordic`, `g721decode` and `g721encode`, its performance degradation becomes very small. This is because those *low*-utility benchmarks do not have high demand for cache resources, thus leaving enough cache space for `vortex`. In contrast, when `vortex` is executed simultaneously with *high*-utility and *saturating*-utility applications, its performance degrades substantially owning to the excessive interferences from the real-time thread.

Figure 3 compares the performance of different schemes for the high-priority threads, which are normalized to the execution cycles of the base scheme. For all the eight pairs, we find that the prioritized cache achieves the same performance as the base, and the prioritized-partitioned cache achieves the same performance as the partitioned cache. Also, for all the eight pairs except `des-vortex`, both the partitioned and PP caches attain performance very close to the prioritized cache as well. This is because for all these eight MediaBench applications except `des`, a 16K cache performs almost the same as a 32K cache, because of either *low* utility, *saturating* utility and/or insignificant number of L2 misses to noticeably impact performance. By comparison, since the performance of `des` can substantially benefit from larger caches (i.e., from 16K to 32K), both the partitioned and PP caches result in much lower performance than the prioritized cache for `des`. Therefore, for real-time threads that are memory-intensive and have stringent deadline constraints, the prioritized cache is superior to both the partitioned and PP caches. On the other hand, for real-time threads that are insensitive to cache resources, it may be acceptable to use partitioned or PP caches, as long as they can meet the deadline requirement.

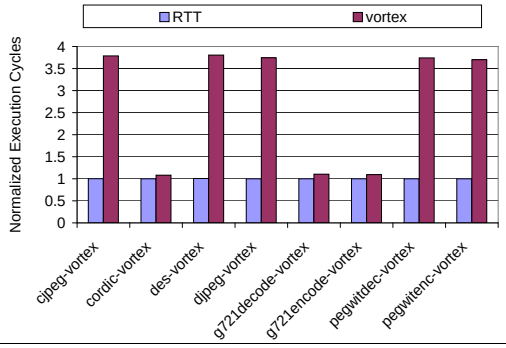


Figure 2. Performance of the prioritized cache, which is normalized to the execution cycles of the base scheme. Note RTT represents the real-time thread of each pair co-running on the dual-core. The cache configuration is based on the default parameters.

Figure 4 shows the performance of the low-priority thread (i.e., `vortex`) with different caches, which are normalized to the execution cycles of the base scheme. As we can see, while the prioritized cache can guarantee the best performance for real-time threads as demonstrated in Figure 3, it also leads to inferior performance for `vortex` in many pairs. In contrast, both the partitioned and PP caches achieve much better performance than the prioritized cache for `vortex`. In particular, we observe that the PP cache outperforms the partitioned cache for all the pairs. The reason is that in a PP cache, the low-priority thread can not only use its own partitioned cache space, but also exploit the spare cache lines that are not used by the (*low/saturating*-utility) high-priority thread. On average, we find that the PP cache can achieve performance 38.4% better than the partitioned cache for `vortex`. Considering the fact that both the PP and partitioned cache have the same performance for real-time threads, we believe the PP cache is a better design option to balance time predictability and performance.

4.2 WCET Results

Table 2 compares the observed WCET (through simulation) and estimated WCET (by performing static timing analysis) for the baseline dual-core processor with a prioritized L2 cache. As can be seen in the last column of Table 2, the estimated WCET is not too far from the observed WCET for most benchmarks, especially considering the fact that the difference between the actual WCET and the observed WCET of a dual-core processor with two-level memory hierarchy is unknown, which indicates superior time predictability of the prioritized cache. The overestimation in our WCET analysis mainly comes from three sources. First, the worst-case execution counts of basic blocks estimated through ILP calculation are often larger than the actual execution counts during simulation (which is a drawback of Chronos itself [4]). Second, the cache static analysis approach [6] used for

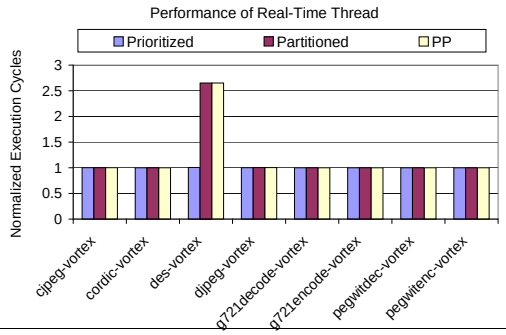


Figure 3. Performance of prioritized, partitioned, and prioritized-partitioned (PP) caches of real-time threads, which is normalized to the execution cycles of the base scheme. The cache configuration is based on the default parameters.

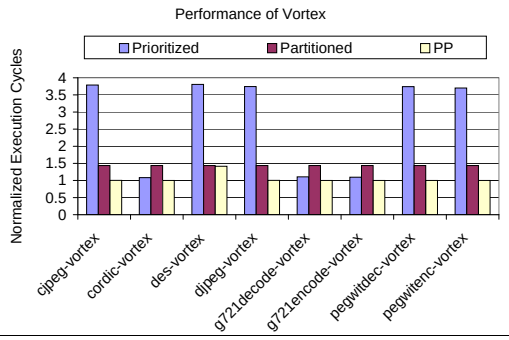


Figure 4. Performance of prioritized, partitioned, and prioritized-partitioned (PP) caches of the non-real-time thread, i.e., `vortex`, which is normalized to the execution cycles of the base scheme. The cache configuration is based on the default parameters.

Benchmarks	Obs. WCET	Est. WCET	Ratio
cjpeg	11356406	11862326	1.045
cordic	1647858	1652098	1.003
des	17920830	21261210	1.186
djpeg	3975849	4331659	1.089
g721decode	367741834	420036834	1.142
g721encode	333650084	371553054	1.114
pegwitdec	25736521	32451811	1.261
pegwitenc	98186680	168297880	1.714

Table 2. Estimated WCET and simulated performance.

the L1 instruction cache analysis is very conservative, which not only directly increases the estimated WCET, but also leads to overestimation of L2 misses. Finally, because the miss latency of L2 is much larger than that of L1, even slight overestimation of L2 misses may have result in large overestimation of the WCET.

5. Conclusions

This paper proposes prioritized caches and prioritized-partitioned caches for achieving time predictability for multi-core processors. Our experiments indicate that prioritized caches lead to good time predictability; however, the performance of the low-priority thread may become worse, especially when the high-priority thread is memory-demanding. Also, our evaluation shows that the prioritized-partitioned cache is superior to the pure partitioned cache, because it allows low-priority threads to efficiently exploit unused or under-utilized cache space of the high-priority threads for performance improvement. We believe the proposed time-predictable cache architectures can provide interesting cache design options to enable real-time multi-core computing.

In our future work, we would like to explore multi-level prioritized caches for multiple real-time tasks with different priorities. Also, we would like to integrate time-predictable caches with multicore real-time scheduling to achieve better schedulability and performance.

Acknowledgment

This work was funded in part by the NSF grants CNS 0720502 and CCF 0914543. We would like to thank anonymous reviewers for their comments to improve the paper.

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckman, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenstrom. The Worst-case execution time problem - overview of methods and survey of tools. In ACM Transactions on Embedded Computing Systems, January 2007.

[2] L. Thiele and R. Wilhelm. Design for time-predictability. In Proc. of the Perspectives Workshop: Design of Systems with Predictable Behavior, April 2004.

[3] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In Proc. of the 18th ACM International Conference on Supercomputing, 2004.

[4] Homepage of Chronos, National University of Singapore. http://www.comp.nus.edu.sg/~rpembed/chronos/.

[5] G. E. Suh, S. Devadas and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In Proc. of International Symposium on High Performance Computer Architecture, 2002.

[6] C. Ferdinand and R. Wilhelm. Fast and efficient cache behavior prediction for real-time systems. In Real-Time Systems, 17((2/3), 1999.

[7] Y. Tan and V. Mooney. A prioritized cache for multi-tasking real-time systems. In Proc. of SASIMI, 2003.

[8] K. Nesbit, J. Laudon and J. Smith. Virtual Private Caches. In Proc. of ISCA, 2007.

A Synchronous Transition Protocol with Periodicity for Global Scheduling of Multimode Real-Time Systems on Multiprocessors

Vincent Nelis[1]
Computer Science Department
Université Libre de Bruxelles (U.L.B.)
Brussels, Belgium
vnelis@ulb.ac.be

Björn Andersson
CISTER Research unit
Polytechnic Institute of Porto
Porto, Portugal
bandersson@dei.isep.ipp.pt

Joël Goossens
Computer Science Department
Université Libre de Bruxelles (U.L.B.)
Brussels, Belgium
joel.goossens@ulb.ac.be

***Abstract*—We consider the global scheduling problem of multimode real-time systems upon identical multiprocessor platforms. During the execution of a multimode system, the system can change from one mode to another such that the current task set is replaced with a new task set. Thereby, ensuring that deadlines are met requires not only that a schedulability test is performed on tasks in each mode but also that (i) a protocol for transitioning from one mode to another is specified and (ii) a schedulability test for each transition is performed. In this paper, we extend the synchronous transition protocol SM-MSO in order to take into account mode-independent tasks, i.e., tasks of which the execution pattern must not be jeopardized by the mode changes.**

***Keywords*-multimode scheduling; multiprocessor scheduling; real-time scheduling;**

I. Introduction

Hard real-time systems require both functionally correct executions and *results that are produced on time*. Currently, numerous techniques exist that enable engineers to design real-time systems while guaranteeing that all their temporal requirements are met. These techniques generally model each functionality of the system by a *recurrent* task, characterized by a computing requirement, a temporal deadline and an activation rate. Commonly, real-time systems are modeled as a set of such tasks. However, some applications exhibit multiple behaviors issued from several operating modes (e.g., an initialization mode, an emergency mode, a fault recovery mode, etc.), where each mode is characterized by its own set of functionalities, i.e., its set of tasks. During the execution of such *multimode* real-time systems, switching from the current mode (called the *old-mode*) to another one (the *new-mode* hereafter) requires to substitute the current executing task set with the set of tasks of the new-mode. This substitution introduces a transient stage, where the tasks of the old- and new-mode may be scheduled *simultaneously*, thereby leading to an *overload* which can compromise the system schedulability.

[1]Supported by the Belgian National Science Foundation (F.N.R.S.) under a F.R.I.A. grant.

The scheduling problem during a transition between two modes has multiple aspects, depending on the behavior and requirements of the old- and new-mode tasks when a mode change is initiated (see e.g., [2], [3] for details about the different task requirements during mode transitions). For instance, an *old-mode task* may be immediately aborted, or it may require to complete the execution of its current instance (in order to preserve data consistency for instance). On the other hand, a *new-mode task* sometimes requires to be activated as soon as possible, or it may also have to delay its first activation until all the tasks of the old-mode are completed. Moreover, there may be some tasks (called *mode-independent* tasks) present in both the old- and new-mode, such that their periodic (or sporadic) execution pattern must not be jeopardized by the mode change in progress (such tasks are typically daemon functionalities). In the literature (see [4] for instance), a transition protocol is said to be *synchronous* if it does not schedule old- and new-mode tasks simultaneously, otherwise it is said to be *asynchronous*. Furthermore, a synchronous/asynchronous protocol is said to be *with periodicity* if it is able to deal with mode-independent tasks, otherwise it is said to be *without periodicity*.

Related work. Numerous scheduling protocols have already been proposed in the *uni*processor case to ensure the transition between modes (see [4] for a survey of the literature about this uniprocessor problem). Targeting *multi*processor environments, previous work [1] proposed two protocols *without periodicity*: a synchronous protocol called SM-MSO and an asynchronous one called AM-MSO. To the best of our knowledge, these two protocols are the only ones to be proposed for the multimode scheduling problem upon *multi*processor platforms.

This research. In this paper, we extend the protocols SM-MSO proposed in [1] to make it "*with periodicity*". We take into account the mode-independent tasks and we rewrite the validity test of SM-MSO in order to ensure that all the requirements are met during every mode transition.

However this research is a first step since we only consider synchronous protocols. Notice that in this document, we assume that every operating mode of the system is scheduled by a *global*, *preemptive*, *work-conserving* and *fixed job-level priority* scheduling algorithm.

II. Model of computation

A. System and platform specifications

We consider multiprocessor platforms composed of a known and fixed number m of *identical* processors $\{P_1, P_2, \ldots, P_m\}$ upon which a multimode real-time system is executed. "Identical" means that all the processors have the same profile (in term of consumption, computational capabilities, etc.) and are interchangeable.

We define a multimode real-time system τ as a set of x operating modes noted $M^1, M^2, \ldots, M^x$ where each mode contains its own set of functionalities to execute. At any time during its execution, the system runs in only one of its modes, i.e., it executes only the set of tasks associated with the selected mode, or the system switches from one mode to another one. A mode M^k contains a set τ^k of n_k functionalities denoted $\left\{\tau_1^k, \tau_2^k, \ldots, \tau_{n_k}^k\right\}$. Every functionality τ_i^k is modeled as a *sporadic constrained-deadline* task characterized by three parameters (C_i^k, D_i^k, T_i^k) – a worst-case execution time C_i^k, a minimum inter-arrival separation T_i^k and a relative deadline $D_i^k \leq T_i^k$ – with the interpretation that, during the execution of the mode M^k, the task τ_i^k generates successive *jobs* $\tau_{i,j}^k$ (with $j = 1, \ldots, \infty$) arriving at times $a_{i,j}^k$ such that $a_{i,j}^k \geq a_{i,j-1}^k + T_i^k$ (with $a_{i,1}^k \geq 0$), each such job has an execution requirement of at most C_i^k, and must be completed at (or before) its absolute deadline denoted $d_{i,j}^k \stackrel{\text{def}}{=} a_{i,j}^k + d_i^k$. In our study, all the tasks are assumed to be independent, i.e., there is no communication, no precedence constraint and no shared resource (except the processors) between them.

At any time t during the system execution, a job $\tau_{i,j}^k$ is said to be *active* iff $a_{i,j}^k \leq t$ and it is not completed yet. Hereafter, $\mathrm{active}(\tau^k, t)$ denotes the subsets of active tasks of τ^k at time t. A task must be *enabled* to generate jobs, and the system is said to run in mode M^k only if every task of τ^k is enabled and all the tasks of the other modes are disabled. Thereby, disabling a task τ_i^k prevents future job arrivals from τ_i^k. In the following, we denote by $\mathrm{enabled}(\tau^k, t)$ and $\mathrm{disabled}(\tau^k, t)$ the subsets of *enabled* and *disabled* tasks of τ^k at time t, respectively.

During any mode change from mode M^i to mode M^j, we denote by $\tau_{i,j}^{\mathrm{mit}}$ the set of mode-independent tasks that belong to both modes M_i and M_j (i.e., $\tau_{i,j}^{\mathrm{mit}} = \tau^i \bigcap \tau^j$). These tasks are assumed to be sporadic and constrained-deadline. Each such task sporadically generates jobs during the entire mode transition and its sporadic execution pattern must not be influenced by the mode change in progress.

B. Scheduler specifications

We consider in this study that the scheduler is *global*, *preemptive*, *work-conserving* and it assigns *fixed job-level priority* according to the usual interpretations (see [1] for formal definitions). Notice that *Global Deadline Monotonic* and *Global Earliest Deadline First* [5] are some examples of such scheduling algorithms. We assume that every mode M^k of the system uses its own scheduling algorithm noted S^k and the tasks set τ^k of every mode M^k can be scheduled by S^k on m processors without missing any deadline. This assumption allows us to only focus on the schedulability of the system *during the mode transitions*, and not during the executions of the modes.

C. Mode transition specifications

While the system is running in a mode M^i (i.e., the old-mode), a mode change can be initiated by any task of τ^i or by the system itself, whenever it detects a change in the environment or in its internal state. This is performed by invoking a $\mathrm{MCR}(j)$ (i.e., a Mode Change Request), where M^j is the destination mode (i.e., the new-mode). We denote by $t_{\mathrm{MCR}(j)}$ the invoking time of a $\mathrm{MCR}(j)$ and we assume that a MCR may only be invoked in the steady state of the system, and not during the transition between two modes.

Suppose that the system is running in mode M^i and a $\mathrm{MCR}(j)$ is invoked (with $j \neq i$). At time $t_{\mathrm{MCR}(j)}$, the system entrusts the scheduling decisions to a transition protocol. This protocol *immediately* disables all the old-mode tasks that are not mode-independent (i.e., the tasks of $\tau^i \setminus \tau_{i,j}^{\mathrm{mit}}$), *hence preventing new job arrivals from these tasks*. At time $t_{\mathrm{MCR}(j)}$ the active jobs of these disabled tasks, henceforth called the *rem-jobs* (for *rem*aining *jobs*), may have two distinct behaviors: either *they can be aborted* or *they must complete their execution*. In previous work [1] we showed that aborting rem-jobs immediately do not jeopardize the system schedulability, that scheduling problem is straightforward. In this research we consider the more interesting case where all rem-jobs *must complete their execution*.

By assumption, we know that the set τ^j of new-mode tasks can be scheduled upon the m processors without missing any deadline. However, the rem-jobs may cause an overload if the tasks of τ^j are immediately enabled upon the mode change request $\mathrm{MCR}(j)$. As a result, transition protocols usually have to delay the enablement of these new-mode tasks until it is safe to do that. We denote by $\mathcal{D}_k^j(M^i)$ *the relative enablement deadline of the task* τ_k^j *during the transition from the mode* M^i *to the mode* M^j, with the following interpretation: the transition protocol must ensure that τ_k^j is enabled not after time $t_{\mathrm{MCR}(j)} + \mathcal{D}_k^j(M^i)$. The goal of a transition protocol is therefore to *(i) complete every rem-job, (ii) schedule every mode-independent tasks and (iii) enable every task of the new-mode* M^j*, while meeting all the job and enablement*

deadlines. When all the rem-jobs are completed and all the tasks of τ^j are enabled, the system entrusts the scheduling decisions to the scheduler S^j of the new-mode M^j and the transition phase ends.

III. The protocol SM-MSO

In this section, we present how the *synchronous* protocol SM-MSO proposed in [1] can be extended while considering mode-independent tasks. Notice that we do not consider asynchronous protocols in this document. The main idea of this extension is the following: *upon a* $\mathrm{MCR}(j)$*, every non-mode-independent task of the old-mode (say* M^i*) is disabled and both the rem-jobs and the mode-independent tasks continue to be scheduled by* S^i *upon the* m *processors. When* all *the rem-jobs are completed, all the non-mode-independent new-mode tasks (i.e., the tasks of* $\tau^j \setminus \tau^{\mathrm{mit}}_{i,j}$*) are simultaneously enabled*. From this instant, both the new-mode tasks and the mode-independent tasks are scheduled by S^j upon the m processors. Figure 1 depicts an example with a 2-processors platform. Both modes M^i and M^j contain 3 tasks and 1 mode-independent tasks ($\tau^{\mathrm{mit}}_{i,j} = \{\tau_1\}$), where the light gray, dark gray and black boxes are the old-mode, new-mode, and mode-independent tasks, respectively. Algorithm 1 gives the pseudo-code of this protocol.

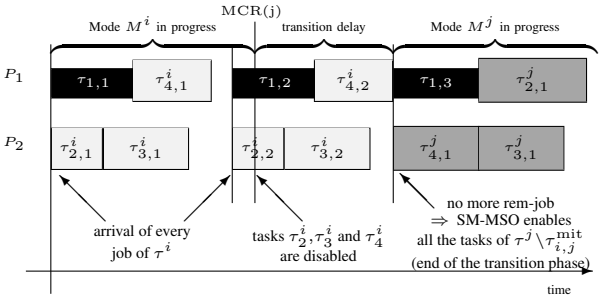


Figure 1. Illustration of a mode transition handled by SM-MSO.

Algorithm 1: SM-MSO (revisited)

Input: $t_{\mathrm{MCR}(j)}$: current time; M^i: the old mode; M^j: the new-mode
begin
  Disable all the tasks $\tau_k \in \tau^i \setminus \tau^{\mathrm{mit}}_{i,j}$;
  Schedule all the jobs of τ^i according to S^i ;
  At job completion time t :
  if $\mathrm{active}(\tau^i \setminus \tau^{\mathrm{mit}}_{i,j}, t) = \phi$ **then**
    enable all the tasks of $\tau^j \setminus \tau^{\mathrm{mit}}_{i,j}$;
    enter mode M^j ;
end

It is well-known that proposing a new scheduling algorithm requires to also provide an associated *schedulability test*, i.e., a condition based on the tasks and platform characteristics which indicates *a priori* whether the given system will meet *every job deadline*. In a similar way, proposing a new mode transition protocol requires to also provide an associated *validity test*, i.e., a condition based on the tasks and platform characteristics that indicates *a priori* whether the given system will meet *every job and enablement deadline during every transition between every pair of operating modes of the system*. In the following, we focus on designing a *validity test* for the protocol SM-MSO. First, Lemma 1 establishes an upper bound on the completion time of any rem-job during any mode transition, while considering the interference due to the execution of the mode-independent tasks. Then in Corollary 1, we determine the *largest makespan*, where the makespan is defined as follows.

Definition 1 (makespan): Let $J = \{J_1, J_2, \ldots, J_n\}$ be a set of n jobs with processing times $c_1, c_2, \ldots, c_n$ that are ready for execution at time 0. Let τ^{mit} be a set of sporadic constrained-deadline tasks that are scheduled during the schedule of J. Suppose that τ^{mit} and the n jobs of J are scheduled upon m identical processors by a global, preemptive, work-conserving and fixed-job priority scheduler. We define the makespan as *the earliest time in the schedule at which all the jobs of* J *are completed*.

By using a similar proof as Lemma 2 of [1], we can easily show that every rem-job and every job generated by any mode-independent tasks always meets its absolute deadline $d_{i,j}$ while using SM-MSO during the transition phases. Thereby, for a given multimode real-time system, the protocol SM-MSO will comply with every expected requirement if all the *enablement deadlines* $\mathcal{D}^j_k(M^i)$ are also met during every mode transition, i.e., if *the makespan is not larger than the minimal enablement deadline of the non-mode-independent new-mode tasks*. This upper bound on the makespan thus allows us to design a *sufficient validity test* that indicates, a priori, if all the enablement deadlines will be met during all possible mode changes. Notice that we do not assume specific scheduling algorithms in this document. Every result proposed here hold for any global, preemptive, work-conserving and fixed-job priority scheduler.

Definition 2 (processed work): At any time t in any schedule, the processed work $\mathrm{w}_k(t)$ denotes the amount of work executed on processor P_k in the time interval $[0, t]$.

Lemma 1: Let $J = \{J_1, J_2, \ldots, J_n\}$ be a set of n jobs with processing times $c_1, c_2, \ldots, c_n$ that are ready for execution at time 0. Let τ^{mit} be a set of sporadic constrained-deadline tasks that are scheduled during the schedule of J. Suppose that τ^{mit} and the n jobs of J are scheduled upon m identical processors by a global, preemptive, work-conserving and fixed-job priority scheduler. Then, an upper

bound $\hat{R}_i$ on the time at which job $J_i \in J$ completes is given by the first fixed point of the following iterative process:

$$\begin{aligned} \hat{R}_i^{(0)} &\leftarrow \frac{1}{m}\sum_{\substack{j=1\\ j\neq i}}^{n} c_j + c_i \\ \hat{R}_i^{(k+1)} &\leftarrow \frac{1}{m}\left(\sum_{\substack{j=1\\ j\neq i}}^{n} c_j + \sum_{\tau_j \in \tau^{\text{mit}}} W(\tau_j, \hat{R}_i^{(k)})\right) + c_i \quad (1) \end{aligned}$$

where $W(\tau_j, t)$ denotes an upper bound on the amount of work that can be generated by the task τ_j in the time interval $[0, t]$. Notice that in [6], the authors showed that

$$W(\tau_j, t) = N_j(t)C_j + \min(C_j, t + D_j - C_j - N_j(t)T_j)$$

where $N_j(t) \stackrel{\text{def}}{=} \left\lfloor \frac{t+D_j-C_j}{T_j} \right\rfloor$

Proof: Suppose that the job J_i completes at time R_i on processor P_k. Since the scheduler is work-conserving, the processed work $\mathrm{w}_k(R_i)$ on P_k is R_i and the processed work $\mathrm{w}_j(R_i)$ on the other processors P_j (with $j \neq k$) is at least $R_i - c_i$. Formally we have

$$\begin{cases} \mathrm{w}_j(R_i) = R_i & \text{if } j = k \\ \mathrm{w}_j(R_i) \geq R_i - c_i & \forall j = 1, \ldots, m \text{ and } j \neq k \end{cases}$$

By summing these m expressions, we get

$$\sum_{j=1}^{m} \mathrm{w}_j(R_i) \geq mR_i - (m-1)c_i \quad (2)$$

where $\sum_{j=1}^{m} \mathrm{w}_j(R_i)$ denotes the amount of work executed on the m processors from time 0 to time R_i. On the other hand, we know that this cumulative processed work cannot be larger than the amount of requested work in the system, i.e.,

$$\sum_{j=1}^{m} \mathrm{w}_j(R_i) \leq \sum_{j=1}^{n} c_j + \sum_{\tau_j \in \tau^{\text{mit}}} W(\tau_j, R_i) \quad (3)$$

By using Inequalities 2 and 3, we get

$$mR_i - (m-1)c_i \leq \sum_{j=1}^{n} c_j + \sum_{\tau_j \in \tau^{\text{mit}}} W(\tau_j, R_i)$$

Rewriting this yields

$$R_i \leq \frac{1}{m}\left(\sum_{\substack{j=1\\ j\neq i}}^{n} c_j + \sum_{\tau_j \in \tau^{\text{mit}}} W(\tau_j, R_i)\right) + c_i$$

As a result, R_i is upper bounded by $\hat{R}_i$ defined as in Expression 1.

■

Corollary 1: Assuming the same notations as in Lemma 1, an upper bound on the makespan is given by

$$\widehat{\mathrm{ms}}(J, \tau^{\text{mit}}, m) \stackrel{\text{def}}{=} \max_{i=1}^{n}\{\hat{R}_i\} \quad (4)$$

The proof is a direct consequence of Lemma 1. As a result, a *sufficient* validity condition may be formalized as follows.

Validity test 1: For any multimode real-time system τ, SM-MSO meets every job and enablement deadline during every transition between every pair of operating modes of τ if, $\forall M^i, M^j$ with $M^i \neq M^j$,

$$\widehat{\mathrm{ms}}(J, \tau_{i,j}^{\text{mit}}, m) \leq \min_{\tau_k^j \in \tau^j \setminus \tau_{i,j}^{\text{mit}}} \{\mathcal{D}_k^j(M^i)\}$$

where J is the set of jobs composed of one job issued from each task of $\tau^i \setminus \tau_{i,j}^{\text{mit}}$.

IV. Conclusion and future work

In this paper, we extended the *synchronous* protocol SM-MSO proposed in [1] in order to take into account the mode-independent tasks during the execution of sporadic multimode real-time systems on multiprocessor platforms. Moreover, we established a validity test which allows the system designer to predict whether the given system can meet all the expected requirements during every mode transition. In our future work, we aim to design an *asynchronous* protocol with the consideration of mode-independent tasks.

References

[1] V. Nelis, J. Goossens, and B. Andersson, "Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms," in *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, Dublin, Ireland, July 2009, pp. 151–160.

[2] G. J. Fohler, "Flexibility in statically scheduled hard real-time systems," Ph.D. dissertation, Technische Universität Wien, 1994.

[3] F. Jahanian, R. Lee, and A. Mok, "Semantics of modechart in real time logic," in *Proceedings of the 21st Hawaii International Conference on Systems Sciences*, 1988, pp. 479–489.

[4] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Real-Time Systems*, vol. 26, no. 2, pp. 161–197, March 2004.

[5] T. Baker, "Multiprocessor EDF and deadline monotonic schedulability analysis," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, December 2003, pp. 120–129.

[6] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, December 2007, pp. 149–160.

Policies for Migration of Real-Time Tasks in Embedded Multi-Core Systems

Kedar M. Katre[1], Harini Ramaprasad[1], Abhik Sarkar[2], Frank Mueller[2]
kedarked@siu.edu, harinir@siu.edu, asarkar@ncsu.edu, mueller@cs.ncsu.edu
[1]*Southern Illinois University Carbondale,* [2]*North Carolina State University*

Abstract

The increasing computational and power demands of embedded systems today are being met by deploying multi-core architectures. Several embedded systems have real-time requirements that necessitate offline temporal guarantees. The use of multicores in such systems poses a challenge in terms of timing predictability, specifically when real-time tasks are permitted to migrate among the different cores.

The aim of this paper is to put forth novel policies to guide migration decisions on time-critical and safety-critical embedded systems that use multicore architectures. Migration decisions are based on the cache usage of tasks, the migration mechanisms available and the characteristics of the network-on-chip (NoC) that is used to provide communication among cores.

1. Introduction

Increasing computational demands over the years have been addressed by increasing the operating clock frequencies of microprocessors as required. However, these designs have reached a clock frequency wall due to area and power considerations, leading to designs with multiple processors on a single chip, known as chip multiprocessors (CMPs) or simply *multicore* processors. The invention of multicore processors has, to a great extent, ensured that the rate of increase in performance of computing systems is maintained, thereby making multicores ubiquitous these days.

The electronics industry has been experiencing an upsurge with the advent of embedded systems. Embedded systems have been a major contributor in reducing the cost, power and area requirements of computing systems. Until recently, embedded systems worked on single-core microprocessors. However, due to increasing computational demands even on such systems, multicore architectures have already found their place in the embedded systems domain.

Prediction of timing behavior to ensure that real-time task deadlines are met is becoming increasingly difficult with the use of multicore platforms in embedded systems. While several real-time multicore scheduling strategies have been and are being proposed to address this issue, their reliance on task migration remains a major challenge. Task migration among cores reduces timing predictability due to cache warm-up overheads while increasing traffic on the Network-on-Chip (NoC) interconnect.

This work was supported in part by NSF grants CNS-0905212, CNS-0905181 and CNS-0720496.

In this paper, we present novel policies to guide migration decisions on embedded multicore systems that require temporal guarantees. Migration decisions are based on cache usage of tasks, the migration mechanisms available and characteristics of the NoC used for communication among cores. We assume that a task may be migrated only between jobs, or, in other words, at the end of the execution of one job and before the next job begins. The core that a task is executing on just before it is migrated is called the *source* core for the migration and the core that the task is migrated to is called the *target* core.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents relevant background information and motivates the problem further. Section 4 describes the methodology used to make migration decisions. Finally, we present our conclusions in Section 5.

2. Related Work

Choffnes *et al.* propose migration policies for multicore fair-share scheduling in the context of soft real-time systems [8]. Their technique minimizes migration costs while ensuring fairness among tasks by maintaining balanced scheduling queues as new tasks are activated. In contrast, our work targets hard real-time systems.

Li *et al.* present migration policies that facilitate efficient operating system scheduling in asymmetric multicore architectures [11], [12]. Their work focuses on fault-and-migrate techniques to handle resource-related faults in heterogeneous cores and does not operate in the context of real-time systems. In contrast, our work focuses on homogeneous cores and strives to improve system utilization by allowing migrations while providing timeliness guarantees for real-time systems.

Yan and Zhang have proposed techniques to calculate the worst-case execution time (WCET) of real-time tasks executing on multicores [19], [18], [20]. Other approaches develop WCET and cache analysis techniques for multi-level caches [14], [10]. All these approaches are limited to shared L2 instruction caches in WCET calculation and they do not consider task migration.

Calandrino *et al.* propose scheduling techniques to account for co-schedulability of tasks with respect to cache behavior [1], [7]. Their approach organizes tasks with the same period into groups of cooperating tasks. While their method improves cache performance in soft real-time systems, they do not specifically address issues related to task migration.

Ramaprasad *et al.* propose techniques to bound the data cache-related preemption delay (D-CRPD) of tasks in the context of periodic, hard real-time systems [15]. We use the results of this work for offline calculation of migration delay bounds (Section 4.1).

3. Motivation and Prior Work

In this section, we present some background information related to task scheduling on multicore systems and further motivate a systematic assessment of migration policies in real-time multicore systems.

3.1. Multicore Scheduling

Scheduling of tasks on cores is an important factor of consideration on multicore systems. Researchers have proposed several schemes for scheduling tasks on multicore systems. They may be broadly classified as partitioned and global scheduling policies.

In *partitioned scheduling* ([9], [6]), tasks are assigned to cores statically and are not allowed to migrate between cores. The advantage of using partitioned scheduling is that there is no migration overhead. However, partitioned scheduling suffers from two main disadvantages. First, such schemes are inflexible and cannot easily accommodate dynamic tasks without a complete re-partition. The re-partitioning problem may be resolved by allocating incoming dynamic tasks to the first available core, but this may not be optimal in terms of overall system utilization. Second, optimal assignment of tasks to cores is an NP-hard problem for which polynomial-time solutions result in sub-optimal partitions.

In *global scheduling* policies, tasks are allowed to migrate between cores as required. Recently, several optimal global scheduling policies have been proposed ([5], [13], [2], [3], [17], [4]). While these schemes strive to overcome the limitations of partitioned scheduling, they add migration overheads to tasks. In the context of real-time systems, the addition of migration overheads changes the timing behavior of tasks, thereby affecting the timing predictability of the system. This necessitates the incorporation of task migration overheads in analysis techniques, thus providing the motivation for, and demonstrating the importance of, the work presented in this paper.

3.2. Reasons for Migration

As discussed in Section 3.1, global real-time scheduling policies on multicores permit task migration among cores. There may be several reasons to do this despite the fact that migration introduces overheads on task execution time. Some of the reasons are listed below.

Capitalize on early task completion: Actual execution time of a real-time task is often significantly less than the worst-case execution time estimate. Early completion of a task on a particular core may be used to an advantage by migrating waiting tasks on a busy core to the newly idle core. This enables earlier start/resumption of the waiting task and improves utilization by minimizing idle time.

Facilitate aperiodic job admission: Task migration may be used to increase admissibility of sporadic jobs (aperiodic jobs with hard deadlines) into the system and to improve response times of aperiodic jobs with soft deadlines.

Avoid costly preemptions: A task potentially preempting a lower-priority task executing on a particular core may be migrated to a different core to allow significant reduction in the preemption delay that would otherwise be incurred by the lower-priority task.

Improve cache performance: If two or more tasks scheduled on the same core overlap significantly in their cache footprint, one or more of them may be migrated to a different core to reduce cache interference.

Balance load, power and thermal characteristics: Task migration may be used to balance the load on cores to ensure that no single core gets overheated while another core is idle.

3.3. Migration Mechanisms

Architectural and hardware support for actual migration may be provided in different ways. In prior work [16], we presented several mechanisms to facilitate task migration among cores. The basics of the mechanisms we use in the current work are described below.

3.3.1. Pull-based model (Conventional warm-up). No specific support for migration is provided in this case. Once the migrated task starts executing on the target core, any cache accesses that result in misses are resolved one at a time using the coherence protocol in effect within the system, either from the shared $L3$ cache or from the $L1$/$L2$ caches of the source core.

3.3.2. Push-based model. In this scheme, cache lines of the task to be migrated are proactively pushed from the source core cache to the target core cache. We currently consider two push models, as described below.

Whole Cache Migration: In this scheme, every line of the source core cache is consulted to identify lines belonging to the task to be migrated that have to be pushed to the target core.

Regional Cache Migration: In this scheme, programmers are allowed to define regions that correspond to a particular cache and only these regions are considered while migrating cache lines.

4. Methodology

In this section, we describe the methodology used to determine when and what task to migrate and where to migrate the task to. For this purpose, we develop a cost-benefit analysis technique that considers several factors to determine the feasibility and usefulness of a given migration.

4.1. Offline Analysis: Migration Delay Bounds

An offline component is employed for calculating the worst-case possible delays introduced into the system due to task migrations. Migration delay bounds include:

- Worst-case Migration Related Preemption Delay (MRPD) experienced by the tasks on the source core due to migration;
- Worst-case MRPD experienced by the existing tasks on the target core due to migration;
- Worst-case Migration Related Cache Delay (MRCD) experienced by the migrated task;
- Worst-case Communication Delay (WCCD) between the source and target cores.

In order to calculate the first three of the four bounds above, we employ static analysis techniques that were developed in prior work to calculate upper bounds on the worst-case cache related preemption delay (CRPD) of tasks [15]. For the calculation of WCCD, we use the worst-case number of hops between the source and target cores and the available network bandwidth as metrics. Further detail about the offline analysis of migration delay bounds is out of the scope of the current paper. Instead, we focus on the online policies to guide migration decisions.

4.2. Online Analysis: Choosing the Best Migration

In Section 3.2, we discussed several reasons that might trigger task migrations. In the current work, we focus on a subset of these triggers. Specifically, migration decisions are made a) when a periodic job is released and b) when a periodic job finishes execution. At the point where a migration decision needs to be made, offline migration delay bounds are first employed to determine whether a possible migration is feasible or not.

A migration is said to be *feasible* if the system remains schedulable in spite of the migration overhead introduced. In other words, a migration is feasible if and only if no task misses its deadline due to the migration. At a given time, there may be more than one feasible migration possible. In this section, we present techniques to choose a suitable migration candidate among a given set of feasible migration candidates. Since we only consider feasible migration candidates for comparison, safety of the system is guaranteed.

4.2.1. Comparing Feasible Migrations: Greedy Approach. Although migration delay bounds are necessary to determine the feasibility of a migration, they are pessimistic bounds due to the fact that they depend entirely on statically available information. In order to identify which of a set of feasible migrations is the least expensive, we present a set of online calculations that may be used to compare feasible migrations.

Migration overheads depend on several metrics. A weighted migration cost for a given migration candidate is calculated based on these metrics. Employing a *greedy* approach, the candidate with the lowest weighted cost is chosen for migration. It is to be noted that the weighted migration cost is a *relative* cost used to compare multiple feasible migrations. In the current work, we make a simplifying assumption that the target core of a migration is empty. In future work, we will consider the effects of a proposed migration on the tasks already allocated to the target core. The metrics considered in this work are described below.

1. Number of Cache Lines. When a task migrates, its cache lines have to be transferred to the intended target core. The worst-case number of cache lines that must be migrated, derived using offline MRCD values for the task under consideration, is a metric used in the calculation of the weighted migration cost.

2. Effect of Migration Mechanism. In Section 3.3, we briefly described the migration mechanisms developed in prior work [16]. The cost of migrating a given task to a specific target core depends on the underlying mechanism that facilitates the migration.

3. Time Until Next Release. As mentioned earlier, in the current work, we only migrate tasks at the end of a job so that the next job may start on the target core. Hence, the time available before the release of the next job of the migrated task is an important consideration while comparing multiple feasible migrations.

4. Distance to Target Core. This metric constitutes the worst-case number of hops between the source and target cores. In the current work, we assume that routes between cores are assigned statically, thereby simplifying the calculation of the number of hops. In future work, we intend to relax this assumption.

5. Quality of Service (QoS). The QoS factor of the network for communication among cores (NoC) is the minimum network bandwidth/latency that is guaranteed to be available along a route at a given point of time. The time taken for transfer of a set of cache lines along a given route is affected by this QoS factor.

4.2.2. Weighted Migration Cost. In this section, we present a method for calculating the weighted cost of a migration relative to other migrations. Equation 1 shows the calculation to determine the overhead of a migration with respect to the release time of the next job of the migration candidate. It is to be noted that migration time that overlaps with the time available before the release of the next job does not count as overhead as far as the response time of the migration candidate is concerned, although it affects the traffic on the NoC.

$$\hat{W}MC_i^{ts_m} = ((nc_i * m_i) * nh_s^t * q) - (t_i^{rel} - ts_m) \quad (1)$$

Here, i is the task number of the migration candidate T_i and ts_m is the start time of the potential migration. nc_i is the worst-case number of cache lines that need to be transferred between the source and target cores and nh_s^t is the worst-case number of hops between the source core (s) and the target core (t). m_i represents the effect of the migration mechanism used. q is the time required for the transfer of one cache line along one hop and represents the global QoS parameter for the NoC bandwidth and latency. t_i^{rel} is the release time of the next job of the migration candidate.

The effect of different migration mechanisms on the weighted migration cost is a comparative term among the mechanisms. It takes into consideration, the advantages and limitations of each mechanism. The less overhead a certain mechanism imposes on the transfer of cache lines between the source and target cores, the lower the value of the factor m_i. The subscript i is used here because the mechanisms affect different tasks in a different manner based on their cache access patterns.

As mentioned in Section 3.3, we assume migration of tasks uses one of three mechanisms, namely 1) Pull-based model (conventional warm-up, CW), 2) Whole Cache Migration (WCM) and 3) Regional Cache Migration (RCM). We also introduce the concept of an *Ideal Migration Mechanism* that assumes the migrated task's cache lines from the source cache are replicated on the target cache with zero overhead. In other words, it is as though there was no migration at all. It is to be noted that this ideal mechanism is not a realistic one, but rather serves as a point of reference for comparison. For ideal migration, we assume that $m_i = 0$ for all tasks. At the other end of the spectrum, we have conventional warm-up that imposes the maximum possible migration overhead since there is no explicit support for migration. For conventional warm-up, we assume that $m_i = 1$ for all tasks. For all other mechanisms, the value of m_i lies in between 0 and 1.

5. Conclusions

This paper presents greedy policies to guide migration decisions on a real-time multicore system. The policy considers the number of cache lines to be migrated, the mechanism being used for migration and characteristics of the NoC to choose the migration with the least overhead at any given time. It is expected that using such a policy to guide migrations will result in reduced response times for tasks and improved overall utilization of the system while guaranteeing real-time deadlines.

References

[1] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 179–190, Apr. 2006.

[2] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Euromicro Conference on Real-Time Systems*, pages 35–43, June 2000.

[3] J. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *Euromicro Conference on Real-Time Systems*, pages 76–85, June 2001.

[4] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *IEEE Real-Time Systems Symposium*, pages 119–128, 2007.

[5] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[6] A. Burchard, J. Liebeherr, Y. Oh, and S. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. on Computers*, 44(12):1429–1442, 1995.

[7] J. Calandrino and J. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Euromicro Conference on Real-Time Systems*, pages 209–308, July 2008.

[8] D. Choffnes, M. Astley, and M. J. Ward. Migration policies for multi-core fair-share scheduling. *ACM SIGOPS Operating Systems Review*, 42:92–93, 2008.

[9] S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[10] D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *IEEE Real-Time Systems Symposium*, Dec. 2008.

[11] T. Li, D. Baumberger, D. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multicore architectures. In *ACM/IEEE Conference on Supercomputing*, Nov. 2007.

[12] T. Li, P. Brett, B. Hohlt, R. Knauerhase, S. McElderry, and S. Hahn. Operating system support for shared-isa asymmetric multi-core architectures. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2008.

[13] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *IEEE Real-Time Systems Symposium*, pages 294–303, Dec. 1999.

[14] F. Mueller. Timing predictions for multi-level caches. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36, June 1997.

[15] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems*, page (accepted), Mar. 2008.

[16] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan. Push-assisted migration of real-time tasks in multi-core processors. In *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 80–89, June 2009.

[17] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *ACM Symposium on Theory of Computing*, pages 189–198, May 2002.

[18] J. Yan and W. Zhang. Time-predictable l2 caches for real-time multi-core processors. In *Work in Progress session of IEEE Real-Time Systems Symposium*, Dec. 2007.

[19] J. Yan and W. Zhang. Wcet analysis of multi-core processors. In *Work in Progress session of IEEE Real-Time Systems Symposium*, Dec. 2007.

[20] J. Yan and W. Zhang. Wcet analysis of multi-core processors with shared l2 instruction caches. In *IEEE Real-Time Embedded Technology and Applications Symposium*, Apr. 2008.

On Optimal Multiprocessor Scheduling Considering Concurrency and Urgency

Jinkyu Lee*, Arvind Easwaran†, Insik Shin* and Insup Lee‡
*Dept. of Computer Science, KAIST, South Korea
†IPP-HURRAY! Research Group, Polytechnic Institute of Porto (ISEP-IPP), Portugal
‡Dept. of Computer and Information Science, University of Pennsylvania, USA
jinkyu@cps.kaist.ac.kr; aen@isep.ipp.pt; insik.shin@cs.kaist.ac.kr; lee@cis.upenn.edu

***Abstract*—It has been widely studied how to schedule real-time tasks on multiprocessor platforms. Several studies find optimal scheduling policies for implicit deadline task systems, but it is hard to understand how each policy utilizes the two important aspects of scheduling real-time tasks on multiprocessors: concurrency and urgency. In this paper, we introduce a new scheduling policy that considers these two properties. We prove that the policy is optimal for the special case when the execution time of all tasks are equally one and deadlines are implicit, and observe that the policy is a new concept in that it is not an instance of Pfair or ERfair. It remains open to prove the optimality of our scheduling policy for general task systems.**

I. Introduction

Real-time schedulability analysis have been studied for achieving predictability on satisfying timing constraints. In particular, scheduling policies for uniprocessor have been extensively studied. EDF [1] and DM [2] are optimal dynamic- and static-priority scheduling policies for preemptive scheduling of periodic and sporadic tasks, respectively. While uniprocessor scheduling has matured over years, finding optimal scheduling policies for general task systems on multiprocessor platforms is still an open problem. Some studies (e.g., [3], [4], [5]) focused on adapting existing uniprocessor scheduling to multiprocessor scheduling, but they do not optimally utilize the processing capacity. This is because uniprocessor policies are not designed to efficiently handle concurrent executions. Several other studies (e.g., [6], [7], [8], [9], [10], [11]) have been introduced that generate optimal schedules for implicit deadline task systems on multiprocessor platforms, but some of these algorithms suffer from high preemption overhead. Further, none of them is able to preserve optimality for more general task systems (such as constrained deadline task systems). We believe this limitation in the state-of-art arises from the fact that it is difficult to understand how existing policies treat some of the important aspects of scheduling real-time tasks on multiprocessor platforms. Therefore, in this paper, we design a novel scheduling policy that clearly differs from existing policies in this regard. That is, it explicitly uses important aspects of scheduling like "job urgency" and "inter-job concurrency" to prioritize jobs.

Handling the trade-off between "job urgency" and "inter-job concurrency" is, in our opinion, the key to efficient scheduling on multiprocessor platforms. This can be explained as follows. To maximize the number of concurrently executing jobs, it is desirable to delay the finishing time of jobs so that more unfinished jobs are available for scheduling. For instance, a policy which gives "higher priority to jobs with longer remaining execution time" implements this concept. However, in order to meet hard real-time requirements it is also important to finish jobs by their deadlines. For instance, a policy which gives "higher priority to jobs with earlier deadline" implements this concept. Therefore, one approach for allowing a trade-off between these concepts would be to simultaneously consider the remaining execution time and deadline of jobs.

In this work, we first consider a simple and intuitive scheduling policy based on the above discussion (called *Dynamic Density First (DDF)*). The dynamic density of a job is defined as its remaining execution time divided by the time to deadline. Note that this parameter changes continuously over a job's lifetime. DDF assigns a higher priority to a job with a larger dynamic density. However, we observed that such a simple (and in some sense crude) strategy does not offer a very fine-grained trade-off between urgency and concurrency. Therefore, it entails another more refined scheduling strategy.

In this paper, we introduce a new scheduling policy extending DDF. We observe and prove that DDF is an optimal multiprocessor scheduler for implicit deadline tasks, when their execution times are all equally one. DDF is not optimal for general tasks with arbitrary execution times however. Looking at how DDF fails to schedule such general tasks, we observe that some jobs are executed earlier than they should be. Its implication is that an optimal schedule can be obtained from a DDF schedule if we can delay the execution of "some" jobs. Reflecting this, we introduce a new scheduling policy called LADD (Lagging And Dynamic Density). A job is said to be *lagging* if it has a longer remaining execution time when compared to some nominal value (we describe this nominal value later in the paper). In LADD, jobs are classified into two groups: a group of lagging jobs and another group of non-lagging jobs. All jobs in the lagging group have a higher priority than those in the non-lagging group. Further, jobs in the same group are scheduled using DDF policy. LADD favors lagging jobs first and then jobs with higher dynamic density; it essentially delays the execution of non-lagging jobs. Our goal is to investigate whether LADD is an optimal multiprocessor scheduling algorithm. In this paper, we show that for the special case where the execution times of all tasks are one, LADD produces the same (optimal) schedule as the one by DDF. We are currently working on proving the optimality of LADD for general task systems.

The contributions of this paper are as follows: we introduce

a new scheduling policy that considers both concurrency and urgency; we prove the optimality of the policy for a special task system; and we observe that the policy is not an instance of Pfair [6] or ERfair [8].

Task Model. We assume a constrained deadline sporadic task model [12]. In this model, a task τ_i is specified as (T_i, C_i, D_i), where T_i is the minimum separation, C_i is the worst-case execution time requirement, and D_i is the relative deadline. We assume $C_i \leq D_i \leq T_i$. A task τ_i invokes a series of jobs, each separated from its predecessor by at least T_i time units. We assume that a single job of a task cannot be executed in parallel. There are m processors in the system, and we assume that $\sum_{\forall j} \frac{C_j}{D_j} \triangleq D_{sys}$ (named as the system static density) is not more than m.

We use $D_i(t)$ and $C_i(t)$ to denote the remaining time to deadline and the remaining execution time, respectively, of a job of τ_i at time t. The dynamic density of a job of τ_i at t is then specified as $\frac{C_i(t)}{D_i(t)}$, and the system dynamic density is $\sum_j \frac{C_j(t)}{D_j(t)} \triangleq D_{sys}(t)$. We express that a job of τ_i is *active* at t when $C_i(t)$ is non-zero. We denote the number of tasks as n, and the number of active jobs at t as $n(t)$.

We consider quantum-based (discrete) systems, and thus the schedule is also quantum-based.

II. Optimality of DDF for $C_i = 1$

In this section, we prove that we can schedule any task set under the following assumptions: (A1) the task set is scheduled by DDF; (A2) the system static density is not more than m; and (A3) execution times of all tasks are one ($C_i = 1$), and thus remaining execution times of all active jobs are also one ($C_i(t) = 1$, $\forall$ active τ_i at t). First, we prove that the system dynamic density does not increase in case of no new arrival of jobs. Second, we prove that the system dynamic density at any time t cannot exceed m in spite of arrival of new jobs.

The first lemma shows that if remaining time to deadline of all jobs are identical, then the system dynamic density cannot increase in case of no new arrival of jobs.

Lemma 1: Assume that the following conditions: (A4) remaining time to deadline of all active jobs at t_0 are identical (*i.e.*, $D_i(t_0) = D(t_0)$); (A5) the system dynamic density at t_0 is not more than m; and (A6) there is no new arrival of jobs in the interval $[t_0, t_1)$. Then, $D_{sys}(t) \leq D_{sys}(t_0)$ for all $t \in [t_0, t_1)$.

Proof: We use mathematical induction.

(The basis) At t_0, $D_{sys}(t_0) \leq D_{sys}(t_0)$.

(The inductive step) We wish to prove the following: if $D_{sys}(t) \leq m$ is true, then $D_{sys}(t+1) \leq D_{sys}(t)$. From (A3) and (A4), we derive $D_{sys}(t) = \frac{n(t)}{D(t)}$ implying $n(t) = D_{sys}(t) \cdot D(t)$. Assuming m jobs are serviced in $[t, t+1)$, we calculate $n(t+1) = n(t) - m$. We then have the results below.

$$D_{sys}(t+1) = \frac{n(t+1)}{D(t+1)} = \frac{D_{sys}(t) \cdot D(t) - m}{D(t) - 1} \leq D_{sys}(t).$$

Here we assumed that m jobs are scheduled in the time interval $[t, t+1)$. If there are less than m active jobs at t, then there is no active job at $t+1$. This means $D_{sys}(t+1) = 0 \leq D_{sys}(t)$, and hence this lemma is always true. ■

We now wish to show that the above lemma holds even when remaining time to deadline of jobs are different. For this purpose, we first show a system dynamic density bounding transformation from a set of jobs with different deadlines to a set of jobs with identical deadline (Lemma 2). Then, in Lemma 3 we prove that Lemma 1 holds even when job deadlines are different.

Lemma 2: Assume $k \cdot \frac{1}{D} = \sum_j \frac{1}{D_j}$ and $D \leq D_j$ for all j. Then, we can derive that $n \cdot \frac{1}{D-t} \geq \sum_{\forall j} \frac{1}{D_j - t}$ for all $0 < t < D$.

Proof:

$$k \cdot \frac{1}{D-t} = k \cdot \frac{1}{D} + k \cdot \frac{1}{D} \cdot \frac{t}{D-t} = \sum_j \frac{1}{D_j} + \sum_j \frac{1}{D_j} \cdot \frac{t}{D-t}$$
$$\geq \sum_j \frac{1}{D_j} + \sum_j \frac{1}{D_j} \cdot \frac{t}{D_j - t} = \sum_j \frac{1}{D_j - t}$$

■

Lemma 3: Assume (A5) shown in Lemma 1, and suppose there is no arrival of new jobs in $[t, t+1)$. Then, we can derive that $D_{sys}(t+1) \leq D_{sys}(t)$.

Proof: Without loss of generality, we sort the index of jobs by remaining time to deadline at t as follows.

$$D_1(t) \leq ... \leq D_m(t) \leq D_{m+1}(t) \leq ... \leq D_{n(t)}(t) \qquad (1)$$

We construct a set of new jobs satisfying the following: (a) the system dynamic density at t of the new jobs is same as that of the original jobs; (b) the m most urgent jobs (as per DDF) from the new set is the same as that in the original set; and (c) other new jobs except the m most urgent jobs are the same as the m^{th} urgent job in the original set. Note that, with this transformation, the number of jobs in the new set at time t is no more than that in the original set. Thus, the set of new jobs can be expressed as follows.

$$D'_1(t) \leq ... \leq D'_m(t) = D'_{m+1}(t) = ... = D'_{n'(t)}(t),$$
$$\text{where } D'_1(t) = D_1(t), ..., D'_m(t) = D_m(t)$$
$$\text{and } \sum_{j=1}^{n'(t)} \frac{1}{D'_j(t)} = \sum_{j=1}^{n(t)} \frac{1}{D_j(t)}, n'(t) \leq n(t) \qquad (2)$$

During $[t, t+1)$ the m most urgent jobs are serviced, so the system dynamic density at $t+1$ of the set described in Eq. (2) is $\sum_{j=m+1}^{n'(t)} \frac{1}{D'_j(t)-1} = \frac{n'(t)-m}{D'_{m+1}(t)-1}$. Here we know that $D'_{m+1}(t)$ is equal to or less than any of $D_{m+1}(t), ..., D_{n(t)}(t)$. By Lemma 2, the system dynamic density at $t+1$ of the set described in Eq. (2) is no less than that of the set described in Eq. (1).

We now define another set of new identical jobs as follows:

$$D_1^*(t) = ... = D_m^*(t) = D_{m+1}^*(t) = ... = D_{n^*(t)}^*(t),$$
$$\text{where } \sum_{j=1}^{n^*(t)} \frac{1}{D_j^*(t)} = \sum_{j=1}^{n(t)} \frac{1}{D_j(t)}, n^*(t) = n'(t) \quad (3)$$

Note these jobs also have the system dynamic density at t same as that of the original job set. Further, it is easy to see that $D_{m+1}^*(t) \leq D'_{m+1}(t)$, and thus the system dynamic density at $t+1$ of the set described in Eq. (3) $\left(\sum_{j=m+1}^{n^*(t)} \frac{1}{D_j^*(t)-1}\right)$ is equal to or larger than that of the set described in Eq. (2) $\left(\sum_{j=m+1}^{n'(t)} \frac{1}{D'_j(t)-1}\right)$. Therefore, by Lemma 1 we get,

$$\sum_{j=1}^{n(t+1)} \frac{1}{D_j(t+1)} \leq \sum_{j=1}^{n'(t+1)} \frac{1}{D'_j(t+1)}$$
$$\leq \sum_{j=1}^{n^*(t+1)} \frac{1}{D_j^*(t+1)} \leq \sum_{j=1}^{n^*(t)} \frac{1}{D_j^*(t)} = \sum_{j=1}^{n(t)} \frac{1}{D_j(t)},$$

and thus we conclude that $D_{sys}(t+1)$ is not more than $D_{sys}(t)$. Similar to Lemma 1, it does not affect the proof that there can be less than m active jobs at t. ■

It now remains to prove that Lemma 1 holds even when new jobs are released in the interval of interest. For this purpose, we first prove that when the remaining time to deadline of a job of τ_i becomes zero, the system dynamic density is at most $m - \left(\frac{1}{D_i}\right)$. The proof technique of the following lemma is similar to that of Lemma 3.

Lemma 4: Assume (A5) in Lemma 1, and suppose there is no arrival of new jobs in $[t, t+D_1(t)]$, where $D_1(t)$ is the remaining time to deadline of the most urgent job at t. We denote the number of most urgent jobs at t as N (note all have deadline at $t+D_1(t)$). Then we conclude $D_{sys}(t+D_1(t)) \leq D_{sys}(t) - \frac{N}{D_1(t)}$.

Proof: Without loss of generality, we sort the index of jobs by the remaining time to deadline at t.

$$D_1(t) = ... = D_N(t) \leq D_{N+1}(t) \leq ... \leq D_{m \cdot D_1(t)}(t)$$
$$\leq D_{m \cdot D_1(t)+1}(t) \leq ... \leq D_{n(t)}(t) \quad (4)$$

We construct a set of new jobs in a similar way to Lemma 3, as follows:

$$D'_1(t) = ... = D'_N(t) \leq D'_{N+1}(t) \leq ... \leq D'_{m \cdot D'_1(t)}(t)$$
$$= D'_{m \cdot D'_1(t)+1}(t) = ... = D'_{n'(t)}(t),$$
$$\text{where } D'_1(t) = D_1(t), ..., D'_{m \cdot D'_1(t)}(t) = D_{m \cdot D_1(t)}(t)$$
$$\text{and } \sum_{j=1}^{n'(t)} \frac{1}{D'_j(t)} = \sum_{j=1}^{n(t)} \frac{1}{D_j(t)}, n'(t) \leq n(t) \quad (5)$$

By Lemma 2, the system dynamic density at $t' \in [t, t+D_1(t)]$ of the set described in Eq. (5) is equal to or larger than that of the set described in Eq. (4).

We now define another set of new identical jobs similar to Lemma 3, but in this case we do not change the N most urgent jobs $\{D_1(t), ..., D_N(t)\}$.

$$D_1^*(t) = ... = D_N^*(t) \leq D_{N+1}^*(t) = ... = D_{m \cdot D_1^*(t)}^*(t)$$
$$= D_{m \cdot D'_1(t)+1}^*(t) = ... = D_{n^*(t)}^*(t),$$
$$\text{where } D_1^*(t) = D_1(t), ..., D_N^*(t) = D_N(t),$$
$$\text{and } \sum_{j=1}^{n^*(t)} \frac{1}{D_j^*(t)} = \sum_{j=1}^{n(t)} \frac{1}{D_j(t)}, n^*(t) = n'(t) \quad (6)$$

We can calculate the system dynamic density at t of the set described in Eq. (6) by $U_{sys}^*(t) = \frac{N}{D_1^*(t)} + \frac{n^*(t)-N}{D_{N+1}^*(t)}$. During $[t, t+D_1^*(t)]$, $D_1^*(t) \cdot m$ jobs are serviced, and thus, using $U_{sys}^*(t)$, we calculate the system dynamic density at $t+D_1(t)$ of this set as follows.

$$U_{sys}^*(t+D_1(t)) = \frac{n^*(t) - D_1^*(t) \cdot m}{D_{N+1}^*(t) - D_1^*(t)} =$$
$$\frac{n^*(t) - D_1^*(t) \cdot U_{sys}^*(t)}{n^*(t) - D_1^*(t) \cdot m} \left(U_{sys}^*(t) - \frac{N}{D_1(t)}\right) \leq U_{sys}^*(t) - \frac{N}{D_1(t)}$$

Using arguments identical to Lemma 3 we can conclude that $D_{sys}(t+D_1(t))$ is equal to or smaller than $D_{sys}(t) - \frac{N}{D_1(t)}$. ■

Using the previous lemmas, we finally have the following theorem.

Theorem 1: DDF can schedule any task set which satisfies (A2) and (A3).

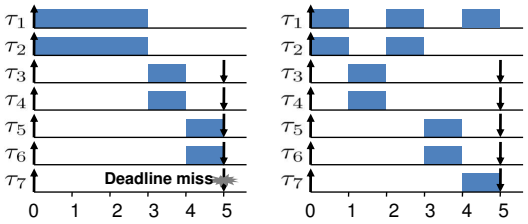
Proof: Assume that the system dynamic density at t is equal to or less than m. By Lemma 4, $D_{sys}(t+D_1(t))$ is not more than $D_{sys}(t) - \frac{N}{D_1(t)}$. At $t+D_1(t)$, we have enough slack in the system dynamic density to accommodate the arrival of a new job of task τ_1. Since we assume constrained deadline tasks, we then guarantee that the arrival of new jobs of τ_1 cannot make the system dynamic density larger than m. Since we know the system dynamic density cannot increase without arrival of new jobs from Lemma 3, it is enough to look at points when remaining time to deadline of any job becomes zero. Since the system dynamic density at the start is not larger than m (system static density is at most m), we then guarantee that the system dynamic density never exceeds m.

At any time, there are at most m urgent jobs (*i.e.*, $C_i(t) = D_i(t)$), and these jobs have the highest priorities. Therefore, DDF can schedule any task set which satisfies (A2) and (A3). ■

A direct corollary of the above theorem is that DDF is optimal for implicit deadline task systems when execution time of all tasks are equally one.

III. Toward Optimality of LADD

We have proved in Section II that DDF is an optimal scheduling policy when the execution times of all tasks are equally one and deadlines are equal to periods. But it can be easily verified that DDF is not optimal without such an assumption on the execution time. Consider a task system that comprises of seven tasks as follows: $\tau_1 = \tau_2 = (14, 7, 14)$, $\tau_3 = \tau_4 = \tau_5 = \tau_6 = \tau_7 = (5, 1, 5)$. This task set is scheduled on a multiprocessor platform that consists of two processors. As shown in Figure 1(a), if we apply DDF, τ_7 cannot be



(a) The original task set (b) The alternative task set

Figure 1. Schedule under DDF

scheduled until its deadline $t = 5$. This scheduling failure comes from early execution of τ_1 and τ_2. If we substitute the original set by a new task set where $\tau_1' = \tau_2' = (2, 1, 2)$ and other tasks are the same, DDF produces a feasible schedule as shown in Figure 1(b). Since any feasible schedule of the new task set can be used for the original task set, we can see that DDF's schedule becomes a feasible schedule by postponing the execution of τ_1 and τ_2.

To improve DDF, we must answer the question "when to delay the execution of jobs and which ones." For this we consider a parameter called the expected remaining execution time of task τ_i (denoted as $C_i^E(t)$). If a job of τ_i is ideally scheduled with a rate of $\frac{C_i}{D_i}$, its remaining execution time at t becomes $C_i^E(t)$, which means $C_i^E(t) = \frac{C_i}{D_i} \cdot D_i(t)$. A job is said to be *lagging* if $C_i(t)$ is strictly larger than $C_i^E(t+1)$. The intuitive meaning of a *lagging* job is that if the job is not serviced in $[t, t+1)$, its remaining execution time becomes larger than its expected remaining execution time at $t+1$. Using this concept of lagging, we now introduce a new scheduling policy called LADD (Lagging And Dynamic Density). In LADD, we divide jobs into two groups: lagging jobs and non-lagging jobs. At every t, we schedule m lagging jobs which have higher dynamic density at t (scheduled by DDF). If there are less than m lagging jobs, we schedule non-lagging jobs also prioritized using DDFThe following theorem proves that LADD and DDF are equivalent for the task system considered in the previous section.

Theorem 2: LADD can schedule any task set, which satisfies (A2) and (A3) shown in Section II.

Proof: Any active jobs at t satisfy the following.

$$C_i^E(t+1) = \frac{C_i}{D_i} \cdot (D_i(t+1)) = \frac{1}{D_i} \cdot (D_i(t) - 1) < 1 = C_i(t),$$

which means any active jobs are lagging. So, LADD produces the same schedule as DDF. By Theorem 1, we conclude that LADD can schedule any task set, which satisfies (A2) and (A3). In other words, LADD is also optimal for any implicit deadline task system where execution time of all tasks are equally one. ■

In the following observation, we claim that LADD is a new scheduling concept.

Observation 1: LADD is not an instance of Pfair or ERfair.

Proof: We provide a example. Consider a task system comprised of six tasks as follows: $\tau_1 = (157, 66, 157)$, $\tau_2 = (667, 174, 667)$, $\tau_3 = (867, 162, 867)$, $\tau_4 = (132, 127, 132)$, $\tau_5 = (878, 120, 878)$, $\tau_6 = (31, 1, 31)$. In this system, $m = 2$ and $D_{sys} < 2$. When we apply LADD, *lag* (as defined in [6]) of τ_5 at $t = 8$ is strictly larger than 1.0. ■

IV. Conclusion

We present a new multiprocessor scheduling policy that offers a fine-grained trade-off between concurrency and urgency. We prove that the proposed scheduling policy is optimal for implicit deadline task systems where execution time of all tasks are equally one. We also observe that our scheduling policy is not an instance of Pfair or ERfair.

Our future work involves deriving a schedulability condition for general task systems under LADD. Another direction of our future work is to find the theoretical bound on the number of preemptions and migrations. We also plan to compare overhead of LADD with that of other scheduling algorithms (*e.g.*, EKG [9] and LLREF [7]) through simulation and/or analysis.

Acknowledgement

This research was supported in part by IT R&D program of MKE/KEIT of Korea [2009-KI002090, Development of Technology Base for Trustworthy Computing], National Research Foundation of Korea (2009-0086964), and KAIST ICC, KIDCS, KMCC, and OLEV grants.

This work was also partially funded by the Portuguese Science and Technology Foundation (Fundação para a Ciência e a Tecnologia - FCT) and the European Commission through grant ArtistDesign ICT-NoE-214373.

References

[1] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] J. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic real-time tasks," *Performance Evaluation*, vol. 2, pp. 237–250, 1982.

[3] S. Cho, S.-K. Lee, S. Ahn, and K.-J. Lin, "Efficient real-time scheduling algorithms for multiprocessor systems," *IEICE Trans. on Communications*, vol. E85–B, no. 12, pp. 2859–2867, 2002.

[4] A. Srinivasan and S. Baruah, "Deadline-based scheduling of periodic task systems on multiprocessors," *Information Processing Letters*, vol. 84, no. 2, pp. 93–98, 2002.

[5] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *RTSS*, 2001.

[6] S. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: a notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[7] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *RTSS*, 2006.

[8] J. H. Anderson and A. Srinivasan, "Early-release fair scheduling," in *ECRTS*, 2000, pp. 35–43.

[9] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *RTCSA*, 2006, pp. 322–334.

[10] K. Funaoka, S. Kato, and N. Yamasaki, "Work-conserving optimal real-time scheduling on multiprocessors," in *ECRTS*, 2008.

[11] B. Andersson and K. Bletsas, "Sporadic multiprocessor scheduling with few preemptions," in *ECRTS*, 2008, pp. 243–252.

[12] S. Baruah, A. Mok, and L. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *RTSS*, 1990.

A Stochastic Framework for Multiprocessor Soft Real-Time Scheduling*

Alex F. Mills
Department of Statistics and Operations Research
University of North Carolina at Chapel Hill

James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

Prior work has shown that the global earliest-deadline-first (GEDF) scheduling algorithm ensures bounded deadline tardiness on multiprocessors with no utilization loss; therefore, GEDF may be a good candidate scheduling algorithm for soft real-time workloads. However, such workloads are often implemented assuming an average-case provisioning, and in prior tardiness-bound derivations for GEDF, worst-case execution costs are assumed. As worst-case costs can be orders of magnitude higher than average-case costs, using a worst-case provisioning may result in significant wasted processing capacity. In this paper, prior tardiness-bound derivations for GEDF are generalized so that execution times are probabilistic, and a bound on expected (mean) tardiness is derived. It is shown that, as long as the total expected utilization is strictly less than the number of available processors, the expected tardiness of every task is bounded under GEDF. The result also implies that any quantile of the tardiness distribution is also bounded.

1 Introduction

The advent of multicore platforms has led to renewed interest in multiprocessor scheduling techniques for real-time workloads. In this paper, we consider an important category of such workloads, namely, those with *soft* timing constraints. The specific such constraint that we consider is that *deadline tardiness* be bounded. We consider this constraint in the context of implicit-deadline sporadic task systems.

In work on such systems, Leontyev and Anderson showed that a variety of global scheduling algorithms are capable of ensuring bounded tardiness without utilization loss [3]. This result extended an earlier proof by Devi and Anderson that showed the same of the *global earliest-deadline-first* (GEDF) scheduling algorithm [1]. Partly because of this result, GEDF is currently being seriously considered for support in real-time Linux [6] (as defined by the REALTIME PREEMPT patch).

While these tardiness-bound results bode favorably for the viability of certain global algorithms like GEDF for soft real-time scheduling, they were established assuming a *worst-case* system provisioning: when specifying a task's utilization, which defines its needed processor share, a worst-case execution time is assumed. This is a serious impediment. Indeed, on a multicore platform, worst-case execution costs could conceivably be orders of magnitude greater than average-case costs, making an average- or near-average-case provisioning all but inevitable in many settings. When using Linux, which lacks the determinism of a real-time operating system, a worst-case provisioning is even more questionable. Additionally, timing-analysis tools have not matured enough to effectively determine reasonably tight upper bounds on task execution times on multiprocessors; on the other hand, mean execution times can be easily estimated in an unbiased way from observed data.

In light of these observations, we argue that, if bounded tardiness is acceptable, then the use of a less conservative task execution model should be as well. The natural model for execution times that vary from job to job is a probability distribution. Motivated by this, we present in this paper a derivation of expected tardiness under GEDF when execution times are stochastic. This derivation generalizes that used in the earlier worst-case bound established by Devi and Anderson [1] and places GEDF on a more solid footing as a candidate soft real-time scheduling algorithm.

Related Work in Queueing. It is natural to think about possible parallels between soft real-time scheduling and queueing. Problems with 'due-date' or 'lead-time' requirements have been examined in the queueing literature from a control standpoint, both in admissions and in service disciplines (see, e.g., [5, 2]). In existing work, a single server is assumed, or an asymptotic regime is used where multiple servers reduce to a single server. In this asymptotic regime, an optimal approach for controlling service is the parameterized *generalized longest queue* (GLQ) rule [5]. In a GLQ(θ) discipline, if $N_i(t)$ is the number of customers in the queue of the ith customer type at time t, then service is provided to the customer type with the largest value of $\theta_i N_i(t)$. The GLQ discipline approximates GEDF under the following conditions:

- θ_i is the mean interarrival time of customer type i;

*Work supported by AT&T, IBM, and Sun Corps.; NSF grants CNS 0834270 and CNS 0834132; ARO grant W911NF-09-1-0535; and AFOSR grant FA9550-09-1-0549.

- customer types are indexed in ascending manner by relative deadline; and
- ties are broken by serving the customer type with the smallest index.

The major difference between a queueing system with several customer types and a sporadic task system is that sporadic tasks are sequential: jobs of a single task must execute in order, and they may not overlap. Such precedence constraints do not generally arise in queueing problems.

Nonetheless, the asymptotic optimality of GLQ suggests that GEDF should be a useful algorithm for scheduling sporadic task systems where execution times are stochastic, but to our knowledge, this issue has not be considered in prior work.

Main Contribution. The main contribution of this paper is an expected tardiness bound under GEDF when task execution costs are defined probabilistically, instead of deterministically; as a byproduct of this analysis, we also obtain bounds on the quantiles (or percentiles) of the tardiness distribution. Note that our analysis is not a simple matter of replacing worst-case execution costs with average-case costs—rather, we assume that execution costs follow a probability distribution, conduct an analysis of the system, and then give a bound on tardiness that results in expectation. This bound depends on the specific execution-time distributions through mean, variance, and worst-case execution time. Thus, our results can be applied without specific knowledge of these distributions. Our bounds are applicable if the *expected* total utilization is less than the system's capacity. In prior work on worst-case tardiness bounds, a similar requirement is needed from a worst-case perspective. However, we allow the system to be *over*-utilized in the worst case.

2 System Specification and Properties

In this section, we formalize the task system to be studied. Throughout this paper, we will denote random variables by capital letters, and deterministic quantities by lowercase letters. All time values are continuous.

Task System. A *task system* τ is a collection of sporadic tasks $\{\tau_i, i = 1 \ldots, n\}$. Each *task* τ_i is a possibly infinite sequence of jobs $\{\tau_{i,j}, j = 1, 2, \ldots\}$. A *job* is a segment of code that requires execution on a processor. Jobs must execute sequentially; that is, the next job of a task cannot begin execution until the previous job of that task has completed. We assume that m processors are available to schedule τ.

A task τ_i is specified by its *period* p_i, and its *execution time distribution function* $G_i(x)$, which gives the probability that a job of τ_i requires no more than x time units to execute. We require such a distribution to have finite mean and variance. Any of the standard probability distributions used for modeling, such as uniform, exponential, Weibull, etc., have this property; however, for the analysis to be correct, we also need an upper bound on the expected value of the maximum execution time seen so far at any point in time—a worst case execution time e_i will suffice. This is the same as assuming that there exists $e_i < \infty$ such that $G_i(e_i) = 1$.

This specification of tasks is very general. For example, the deterministic case often considered in the real-time systems literature, where every task requires exactly its worst-case execution time, is a special case of our system, where $G_i(x)$ is 1 if $x \geq e_i$, and 0 if $x < e_i$.

Schedule-Independent Properties of a Job. The following characteristics of a job $\tau_{i,j}$ do not depend on how jobs are scheduled. The *release time* $r_{i,j}$ is the earliest time that job $\tau_{i,j}$ may execute. We assume the sporadic model, for which $r_{i,1} \geq 0$ and $r_{i,j} \geq r_{i,j-1} + p_i$ for all i, j. The *deadline* $d_{i,j}$ is the time by which $\tau_{i,j}$ must complete execution. We assume implicit deadlines, so $d_{i,j} = r_{i,j} + p_i$. The *execution time* $X_{i,j}$ is the actual time that $\tau_{i,j}$ executes on a processor, so $\mathsf{P}(X_{i,j} \leq x) = G_i(x)$.

Job $\tau_{i,j}$ is *active* at time t if $r_{i,j} \leq t$ and it has not finished executing. A task is active at time t if any of its jobs are active at time t. Job $\tau_{i,j}$ is *eligible* at time t if it is active and its predecessor $\tau_{i,j-1}$, if it exists, is not active.

Each of the above properties depends only on the task specification. We assume that the execution time of $\tau_{i,j}$ is independently and identically distributed according to the distribution $G_i(\cdot)$. This means that the execution of each job of τ_i follows the same distribution, and it does not depend on the execution time of some other job. Note that distributions are not identical across tasks, only jobs of the same task.

Properties of a Task System in Expectation. The *expected execution time* of a job of task τ_i is given by

$$\bar{e}_i = \mathsf{E}(X_{i,1}) = \mathsf{E}(X_{i,2}) = \cdots = \int_0^\infty x dG_i(x).$$

The *expected utilization of a task* τ_i is $\bar{u}_i = \bar{e}_i/p_i$. The *expected total utilization of* τ is therefore $\bar{u}_{\text{sum}} = \sum_i \bar{u}_i$ (for background on properties of expectation, see for example, [4, ch. 7]). The *execution time variance* σ_i^2 of a task τ_i is given by $\mathsf{E}\left(X_{i,1}^2\right) - \bar{e}_i^2$.

Schedule-Dependent Properties of a Job. The following characteristics of a job $\tau_{i,j}$ depend on how jobs are scheduled under scheduler S. The *completion time* $C_{i,j}^{\mathsf{S}}$ is the actual time the job completes executing. The *tardiness* $T_{i,j}^{\mathsf{S}}$ is the amount of time that the job is late:

$$T_{i,j}^{\mathsf{S}} = \max\{C_{i,j}^{\mathsf{S}} - d_{i,j}, 0\}. \tag{1}$$

We will omit the superscript S when it is clear which scheduler is assumed, and we will omit $\max$ from the above expression when it is clear that $T_{i,j}^{\mathsf{S}}$ is non-zero.

Definition (Stability)**.** τ is *stable* if $\bar{u}_{\text{sum}} < m$, where m is the number of processors, and $\bar{u}_i < 1$ for all $\tau_i \in \tau$.

All task systems considered in this paper will be assumed to be stable.

Definition (Schedulability)**.** τ is *schedulable* by a scheduling algorithm S if it can be scheduled by S in such a way that the expected tardiness of every job is bounded.

The following processor-sharing scheduler will be a tool in our analysis.

Definition. PS is a processor-sharing (PS) schedule on m processors where for all i, at every instant that τ_i is active, we allocate to τ_i a fraction $\hat{u}_i$ of the processing capacity of one processor, where

$$\sum_{i=1}^{n} \hat{u}_i \leq m \tag{2}$$

$$\bar{u}_i < \hat{u}_i \leq 1, \quad \forall i. \tag{3}$$

At instants when τ_i is not active, it receives no allocation.

When τ_i is active, the fraction of processing capacity allocated to τ_i is thus strictly greater than the fraction needed for it to complete on time in the average case—note that under this model, some jobs may *not* complete by their deadlines in PS. This is a major difference in comparison to how PS schedules are usually defined.

We can imagine that the PS schedule is a system of n processors, each of which has a fraction $\hat{u}_i$ of the processing capacity of one processor in the real system, and each of which is dedicated to executing jobs of a specific task. It is important to note that there is not a unique choice of values of $\{\hat{u}_i, i = 1, 2, \ldots, n\}$, and the choice of values will affect the expected tardiness bound we derive; however, since the stability assumption guarantees that we will always have excess processing capacity, we are guaranteed to be able to find values of $\{\hat{u}_i, i = 1, 2, \ldots, n\}$ satisfying (2) and (3).

Example. Consider the example of a task system with three tasks and two processors, with the following specifications: $(p_1, \bar{e}_1) = (1, 0.8)$, $(p_2, \bar{e}_2) = (2.1, 0.7)$, $(p_3, \bar{e}_3) = (1.6, 0.4)$. Then a feasible choice for $\{\hat{u}_i, i = 1, 2, 3\}$ is $\{0.95, 0.4, 0.65\}$, and the corresponding PS schedule for one instance of this task system is given in Fig. 1 (actual execution times were randomly generated).

Definition. *Global earliest-deadline first* (GEDF) is a schedule on m processors such that: at each time instant where there are more than m active tasks, the m active tasks whose eligible jobs have the earliest deadlines are each allocated one processor; and at each time instant where there are $k \leq m$ active tasks, each active task is allocated one processor, and $m - k$ processors are idled. In the case of deadline ties, a consistent tie-breaker is used. We assume that ties are broken in favor of the task with the smallest index.

If the set of release times is fixed, GEDF induces the following ordering on jobs.

Definition (Job Ordering)**.** Given a fixed set of release times, $\tau_{i,j} \prec \tau_{k,l}$ if and only if $d_{i,j} < d_{k,l}$, or $d_{i,j} = d_{k,l}$ and $i < k$.

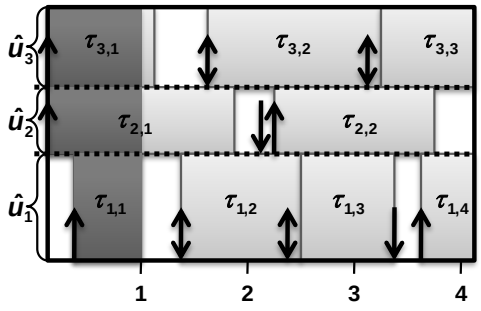


Figure 1: Example PS schedule.

Definition. The *instantaneous schedule* IS is a schedule where at the time a job is released, it instantaneously receives its full allocation.

PS-induced and GEDF-induced Tardiness. The major result of this paper is that stability implies schedulability under GEDF. In the deterministic model of [1], because there is no uncertainty, a stable system can be scheduled using processor sharing with no tardiness. Therefore, tardiness under GEDF resulted only from the use of the non-optimal GEDF as compared to PS.

Our result generalizes [1] for the case where there may be tardiness under PS. In our model, there may be tardiness under PS due to uncertainty of the execution times. We will show that the total tardiness of a job under GEDF has two contributions: PS-*induced tardiness*, which comes from the variability of execution costs, and GEDF-*induced tardiness*, due to using GEDF.

3 Main Result and Proof Sketch

Theorem 1. *Let τ be a stable sporadic task system scheduled using* GEDF*. For any given task $\tau_i \in \tau$, the expected tardiness of each job $\tau_{i,j}$ of task τ_i under* GEDF *schedule is no more than*

$$\beta_i = c + e_i,$$

where c is a constant that depends on the mean, variance, and worst-case execution times of the tasks in τ, and on the feasible choice of $\{\hat{u}_i\}$ used for schedule PS*, and e_i is the worst-case execution time of τ_i.*

Proving this results requires two steps.

GEDF-Induced Tardiness. We first use *a posteriori* analysis to establish the following fact: for each job $\tau_{i,j}$, there an upper bound on the time (denoted by $c_{i,j}$) when it completes in GEDF, expressed in terms of the time (denoted by $f_{i,j}$) when it completes in PS, and the earliest time (denoted by $g_{i,j}$) when $\tau_{i,j}$ *and* all higher priority work according to the ordering $\prec$ completes in PS. We do this through three steps:

1. we show that if some job's completion time does not meet a specified bound in GEDF, then there is a lower bound on how much GEDF 'lags behind' the instantaneous schedule IS, in terms of allocation to jobs of τ,

2. we upper bound how much GEDF 'lags behind' the processor sharing schedule PS, and
3. we show a relationship between the allocation is PS and in IS.

Our *a posteriori* analysis is similar the proof technique of Devi and Anderson [1]. However, the deterministic system specification of [1] results in PS-completion times and job deadlines being identical; in our system specification, they are not. As a result, jobs do not finish in the same order in PS as they do in GEDF. This adds some complexity to the proof as we cannot simply extend parts of the proof that use this assumption. To work around this difference, we use a second ideal system—the instantaneous schedule IS, which is not used in [1].

The result of this analysis is that if we define $\hat{e}_{i,j} = \max_{j' \leq j}\{e_{i,j'}\}$, define $\upsilon = \sum_{\tau_k \in \mathcal{U}_{\max}} \hat{u}_k$, where $\mathcal{U}_{\max}$ is the set of $m-1$ tasks with largest values of $\{\hat{u}_i\}$, and define $\eta_{i,j} = \sum_{\tau_k \in \mathcal{E}_{\max}} \hat{e}_{k,l_k}$, where $\mathcal{E}_{\max}$ is the set of $m-1$ tasks with largest values of $\hat{e}_{k,l_k}$, then

$$c_{i,j} \leq \max\{d_{i,j}, f_{i,j}\} + \frac{\eta_{i,j} + m(g_{i,j} - f_{i,j})}{m - \upsilon} + \hat{e}_{i,j}. \quad (4)$$

Stochastic Analysis of PS-Induced Tardiness. The upper bound in 4 on the GEDF-completion times is related to the tardiness of the job in PS, since $\max\{d_{i,j}, f_{i,j}\}$ is equal to $d_{i,j}$ plus the PS-induced tardiness of $\tau_{i,j}$, and $g_{i,j} - f_{i,j}$ is bounded from above by summing the tardiness of the lowest priority jobs of each task prior to $\tau_{i,j}$. Thus, the second step is to conduct *a priori* analysis on the tardiness of jobs in PS. To prove that the expected (mean) tardiness of each job in PS is bounded, we use a result from queueing to show the existence of a constant ψ so that

$$\mathsf{E}\left(T_{i,j}^{\mathsf{PS}}\right) \leq \hat{u}_i \psi, \quad (5)$$

and we use linear programming to calculate the values of ψ and $\{\hat{u}_i, i = 1, \ldots, n\}$.

By combining (4) and (5), we get the following result:

Lemma 2. *Let τ be a stable sporadic task system. Let $l_k = \max\{l : \tau_{k,l} \prec \tau_{i,j}\}$. Then the expected tardiness of every job $\tau_{i,j}$ in* GEDF *is no more than*

$$\beta_{i,j} = \hat{u}_i \psi + \frac{\eta_{i,j} + m^2 \psi}{m - \upsilon} + \mathsf{E}\left(\hat{e}_{i,j}\right), \quad (6)$$

Theorem 1 follows from this lemma, because there exists c so that $\beta_{i,j} \leq \beta_i$ for all $\tau_{i,j} \in \tau_i$: as long as each task has a worst-case execution time e_i, then $\mathsf{E}\left(\hat{e}_{i,j}\right) \leq e_i$ for all $\tau_{i,j} \in \tau$, and $\eta_{i,j} \leq \sum_{\tau_k \in \mathcal{E'}_{\max}} e_i$ for all $\tau_{i,j}$, where $\mathcal{E'}_{\max}$ is the set of $m-1$ tasks with largest values of e_i.

Finally, the following Corollary follows directly from Markov's inequality [4]:

Corollary 3. *The q-quantile of the tardiness of every job $\tau_{i,j}$ is no more than*

$$\frac{1}{1-q}\beta_i. \quad (7)$$

To summarize the results:

1. there is an upper bound $\beta_{i,j}$ on the expected tardiness of job $\tau_{i,j}$ in GEDF,
2. given the worst-case execution time e_i, there is a constant upper bound β_i on the expected tardiness of any job of τ_i in GEDF, and
3. any quantile (or percentile) of every job's tardiness under GEDF is also bounded by a constant.

4 Conclusion

We have presented a sketch of a probabilistic tardiness-bound derivation for GEDF that is a generalization of the result of [1]. If we use a deterministic model where worst-case execution times are required for every job, then $\bar{e}_i = e_i, \forall i$, $\sigma_i^2 = 0, \forall i$, and then choosing $\psi = 0$, $\hat{u}_i = e_i/p_i, \forall i$ is sufficient for PS-induced tardiness to be zero. Since the system is deterministic, tardiness and expected tardiness are equivalent, resulting in the tardiness bound for any job of τ_k of $\frac{\eta}{m-\upsilon} + e_k$, which almost matches the tardiness bound given in [1], which is $\frac{\eta - e_{\min}}{m-\upsilon} + e_k$, where $e_{\min}$ is the smallest worst-case execution time over all tasks.

This result has practical value because it shows that GEDF can be used to schedule a task system on a multiprocessor, even if the total worst-case utilization exceeds the number of available processors, and even if some tasks' worst-case execution times exceed their periods, provided that total average-case utilization does not exceed the number of processors, and average-case execution times do not exceed periods.

References

[1] U. C. Devi and J. H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, 2005.

[2] B. Doytchinov, J. Lehoczky, and S. Shreve. Real-time queues in heavy traffic with earliest-deadline-first queue discipline. *The Annals of Applied Probability*, 11(2), 2001.

[3] H. Leontyev and J. H. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, 2007.

[4] S. Ross. *A first course in probability*. Prentice Hall, 6 edition, 2002.

[5] J. A. Van Mieghem. Due-date scheduling: Asymptotic optimality of generalized longest queue and generalized largest delay rules. *Operations Research*, 2003.

[6] P. Zijlstra. Deadline scheduling in Linux and why it hasn't happened yet. In *Eleventh Real Time Linux Workshop*, 2009.

An Optimal Scheme for Multiprocessor Task Scheduling: a Machine Learning Approach

Aryabrata Basu, Shelby Funk
University of Georgia
Athens, GA, USA
{abasu,shelby}@cs.uga.edu

***Abstract*—We consider the problem of scheduling periodic task sets on identical multiprocessors. We present NQ-Wrap, an online scheduling algorithm, that can optimally schedule periodic tasks on multiprocessors. We wish to improve this algorithm by reducing the scheduling overhead due to preemptions and migrations. To this end, we introduce a number of rules that can be used to safely modify and NQ-Wrap schedule. We plan to apply these rules by incorporating machine learning techniques.**

I. Introduction

As multiprocessors become more popular, they are used in a wider variety of applications, including real-time and embedded systems. While there are many advantages to using multiprocessor systems, scheduling with these systems can be quite complex. Algorithms that are known to perform very well on uniprocessor systems, such as Earliest Deadline First (EDF)[2] do not perform as well on multiprocessors. To date, optimal multiprocessor scheduling algorithms tend to have restrictions that make them less desirable than some non-optimal algorithms. Some common restrictions are that (i) they have high overhead, (ii) they apply only to a restrictive job model, or (iii) the schedule must be quantum based. But these claims are only restricted to *PFair* family. There are other optimal algorithms like *EKG*[1] and advanced *LLREF* [3].

This paper introduces a set of guidelines and rules that can be used to reduce the number of preemptions and overheads of an optimal online multiprocessor scheduling algorithm. Moreover we will generalize the overhead reduction policy through machine learning approach.

The machine learning approach in general is a broad term and has multifarious components. One of them will be dynamic selection in contrast with scheduling rules. This will be our primary focus in this paper. Dynamic selection of scheduling rules during real operations has been recognized as a promising approach to the scheduling of the production line. For this strategy to work effectively, sufficient knowledge is required to enable prediction of which rule is the best to use under the current line status.

This paper presents a scheduling approach that employs machine learning. Using this technique, while analyzing the earlier performance of the system, scheduling knowledge is obtained whereby the right dispatching rule at each particular moment can be determined. According to Quinlan[7] three different types of machine-learning algorithms can be used to obtain scheduling knowledge: inductive learning, backpropagation neural networks, and case-based reasoning (CBR). Based on the *scheduling knowledge*, a binary decision tree is automatically generated using empirical data obtained by iterative simulations, and it decides online which rule to be used at decision points. But as of now, we will be dealing with *Neural Networks* and the rest of the techniques will be considered in future.

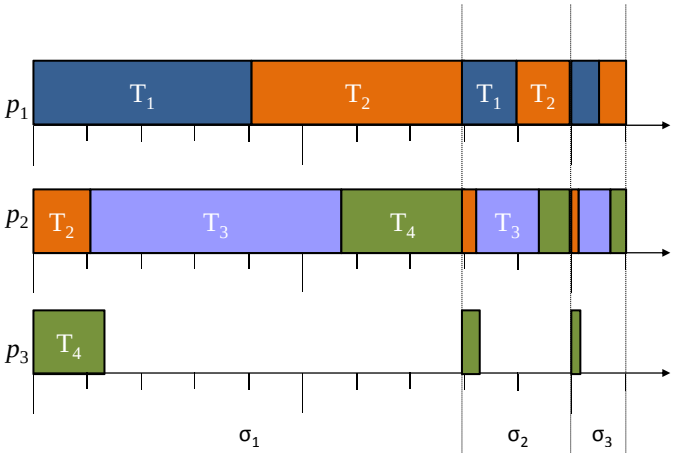


Fig. 1. An illustration of the NQ-Wrap Schedule

II. Model and Definitions

This paper considers the scheduling of *periodic*[5] task sets on multiprocessors. Let, $T_i = (p_i, e_i)$ denote a periodic task. We assume we that tasks are synchronous and deadlines equal to periods. A task T_i is a process that invokes jobs $T_{i,1}, T_{i,2}, \ldots$. Each job has an execution requirement e_i and a relative deadline p_i — if $T_{i,j}$ arrives at time $a_{i,j}$ then it must be allowed to execute for e_i time units during the interval $[a_{i,j}, a_{i,j} + p_i[$. We say $T_{i,j}$'s deadline during this interval is $d_{i,j} = a_{i,j} + p_i$. Each task invokes its first job at time 0 and all the remaining jobs are invoked exactly p_i time units apart — i.e., $a_{i,j} = (j-1)p_i$ for all $j \geq 1$. A task set $\tau = \{T_1, T_2, \ldots, T_n\}$ denotes a set of n periodic or sporadic tasks.

One important parameter used to describe a task with deadlines equal to periods is its utilization $u_i = e_i/p_i$. For periodic tasks, the utilization u_i measures the proportion of

time T_i executes on average. $[u_{max}, U_{sum}]$ are τ's maximum and total utilization, respectively.

NQ-Wrap divides time into consecutive intervals, called *time slices*. The end of each time slice coincides with some job's deadline. We let $\sigma_j = [t_j, t_{j+1}[$ denote the j^{th} time slice, where $t_1 = 0$, and each subsequent t_j is the deadline of some job. The length of σ_j is denoted X_j – i.e., $X_j = t_{j+1} - t_j$.

A task $T_{i,h}$ will overlap with one or more time slices, we let $\Psi_{i,j}$ denote the set of time slices that coincide with $T_{i,h}$'s execution. At all times t, every task T_i has a *local execution requirement*, $\ell_{i,t}$. This is the amount of time that T_i must execute between time t and time $\dot{t}_{j_t}$. A task's *local utilization* is the proportion of time T_i must execute during the remainder of the time slice, namely $r_{i,t} = \ell_{i,t}/(\dot{t}_{j_t} - t)$. A task set's total utilization R_t and total local remaining execution L_t at time t are defined to be the sum of the active task's local utilization and local remaining execution, respectively, viz.,

$$R_t = \sum_{T_i \in Active(t)} r_{i,t} \text{ and } L_t = \sum_{T_i \in Active(t)} \ell_{i,t}$$

We consider the problem of scheduling periodic tasks on identical multiprocessors. Throughout this paper, we let m denote the number of processors. Without loss of generality, we assume the speed of each processor is 1. That is each processor performs one unit of work per unit of time. Henceforth, a "multiprocessor" refers to an identical multiprocessor with m unit speed processors.

III. NQ-Wrap Algorithm

The algorithm NQ-Wrap is based on *time slices*. Each time slice ends at some job's deadline. The algorithm has a global scheduler and a local scheduler for each processor. The local schedulers are table driven. At the beginning of each time slice σ_j, the global scheduler does the following.

1) For each task T_i, calculates $\ell_{i,t_j} = u_i \cdot (t_{j+1} - t_j)$.
2) Puts these executions end-to-end and cuts this long sequence of jobs every X_j time units.
3) "Wraps" these slices around onto the processors – i.e., the first cut is assigned to processor ρ_1, the second cut is assigned to processor ρ_2, and so on.
4) Sends each processor its designated schedule.
5) Sleeps until time t_{j+1}.

This algorithm will successfully schedule any task set on m processors provided $U_{sum} \leq m$ and $u_{max} \leq 1$ (i.e., this is an optimal scheduling algorithm).

Figure 1 illustrates the NQ-Wrap scheduling of the task set $\tau = [T_1, T_2, T_3, T_4]$, where $T_1 = (12, 6)$, $T_2 = (8, 5)$, $T_3 = (10, 6)$, $T_4 = (11, 5)$ for the interval [0,16]. The first five deadlines of τ occur at times 8, 10, 11, 12 and 16 and the periods are 8, 10, 11 and 12. These deadlines define the first five time slices. They are $\sigma_1 = [0, 8[$, $\sigma_2 = [8, 10[$, $\sigma_3 = [10, 11[$, $\sigma_4 = [11, 12[$, and $\sigma_5 = [12, 16[$ as shown above.

In each time slice, the local execution is proportional to the task's utilization. For example, $l_{1,0} = 6/12 \times 8 = 4$, $l_{2,0} = 5/8 \times 8 = 5$, $l_{3,0} = 6/10 \times 8 = 4.8$, and $l_{4,0} = 5/11 \times 8 = 3.6$.

Because step 2 above makes $O(U_{sum})$ cuts, NQ-Wrap invokes $O(U_{sum})$ migrations per time slice. Furthermore, each job in the task set is broken up into pieces. Depending on whether or not each job migrates within time slices, each piece of a job is preempted either once or twice. Our aim is to reduce these expensive operations.

We can reduce preemption and migration overhead by adjusting the local execution of tasks within time slices. We might want to adjust local executions to avoid preemptions. If possible, we would try to concentrate all of a job $T_{i,h}$'s local execution into a single time slice $\sigma_j \in \Psi_{i,h}$ if $X_j \geq e_i$. In this case we might choose to let $T_{i,j}$ execute only in time slice σ_j – i.e., we would set ℓ'_{i,t_j} to e_i and set ℓ'_{i,t_k} to 0 for all other $\sigma_k \in \Psi_{i,h}$. For example we may want to increase $l_{4,0}$ to 5 and let $\ell 4, 8 = \ell_{4,10} = 0$ in Figure 1. In order to do this, other jobs' local executions might have to be reduced in σ_j and increased in other time slices to compensate for $T_{i,j}$'s increased execution (this is not necessary in the example above because there is enough idle time in σ_1 to allow $\ell_{4,0}$ to increase without forcing some other task's local execution to decrease).

Alternatively, we might want to adjust local executions to avoid a migration within some time slice σ_j if a cut splits ℓ_{i,t_j} into 2 pieces of length α_1 (before the cut) and α_2 (after the cut). In this case, we could "give" some of ℓ_{i,t_j} to either ℓ_{i_1,t_j} or ℓ_{i_2,t_j}, where ℓ_{i_1,t_j} is assigned to the same processor as α_1 and ℓ_{i_2,t_j} is assigned to the same processor as α_2. We could either *(i)* increase ℓ_{i_1,t_j} by α_1 and set ℓ_{i,t_j} to α_2 or *(ii)* increase ℓ_{i_2,t_j} by α_2 and set ℓ_{i,t_j} to α_1. For example, we might reduce $\ell_{1,0}$ to 3 in Figure 1, which would keep task T_2 from wrapping to ρ_2 in σ_1. Of course, if we did this then T_1 would have to execute for 3 time units during $[8, 12[$ instead of 2 time units.

Determining the absolute minimum number of preemptions and migrations is NP-complete. In most cases, it will be non-zero. However, it can certainly be reduced significantly from the number invoked by the NQ-Wrap algorithm described above. Our aim is to modify the NQ-Wrap algorithm to incorporate machine learning techniques like *Rule based concept learning* and *Inductive learning scheme*.

These techniques rely on guidelines provided through rules and examples. Below, we present some rules that apply to the NQ-Wrap algorithm. Our goal is to use these rules in conjunction with machine learning techniques to reduce NQ-Wrap's overhead.

IV. Rules

The following must hold:

- For each job $T_{i,h}$ and time slice $\sigma_j \in \Psi_{i,h}$, the local execution cannot exceed the length of the time slice – i.e., $\ell_{i,t_j} \leq X_j$.
- For each time slice σ_j, the total demand cannot exceed available processing time – i.e., $L_{t_j} \leq m \cdot X_j$.
- For each job $T_{i,h}$ it must be allowed to execute for e_i time units between its arrival and deadline – i.e, $\sum_{\sigma_j \in \Psi_{i,h}} \ell_{i,t_j} = e_i$.

One possible approach would be to first increase L_{t_j} to be as large as possible within some time slices and then (perhaps) "swap" execution times of tasks between time slices.

A. Increasing L_{t_j}.

The local execution of tasks can be increased as much as possible provided the three rules above are not violated. For example we can increase $l_{4,0}$ to 5 in Figure 1.

B. Swapping execution

Swapping execution means increasing the local execution of a job $T_{A,x}$ by some amount a and decreasing the local execution of another job $T_{B,y}$ by the same amount a. While swapping could be done both forward and backward in time, the approaches we envision only swap forward in time. Once the local execution times for a time slice have been determined, they will remain fixed and the next time slice will be considered. We may come up with some other better techniques.

With this is mind, we see 2 possible approaches: Explicit and Implicit Swapping. In explicit swapping, the swap amount a is explicitly designated to apply to two time slices σ_{j_1} and σ_{j_2}. In implicit swapping, the swap amount is specified for one time slice σ_{j_1} and the remaining execution of $T_{A,x}$ and $T_{B,y}$ are adjusted accordingly – the change in remaining execution can be spread among the remaining time slices for the jobs.

Explicit swapping. The implementation of explicit swapping are very straightforward. We make the following adjustments.

- $\ell_{A,t_{j_1}} \leftarrow \ell_{A,t_{j_1}} + a$,
- $\ell_{B,t_{j_1}} \leftarrow \ell_{B,t_{j_1}} - a$,
- $\ell_{A,t_{j_2}} \leftarrow \ell_{A,t_{j_2}} - a$, and
- $\ell_{B,t_{j_2}} \leftarrow \ell_{B,t_{j_2}} + a$.

Of course, once such a swap has occurred, rule 1 will no longer be applied at the beginning of each time slice.

For example, in Figure 1, we have $\ell_{1,8} = 1$ and $\ell_{4,8} = 0.9$. Also, $\ell_{1,10} = 0.5$ and $\ell_{4,10} = 0.4$. We could explicitly swap 0.5 units of work between tasks T_1 and T_4 in time slices σ_2 and σ_3. To do this, we would increase $\ell_{1,8}$ to 1.5 in σ_2, decrease $l_{4,8}$ to 0.4 in σ_2, decrease $l_{1,10}$ to 0 and increase $l_{4,10}$ to 0.9.

Implicit swapping. The implementation of implicit swapping is a little more complicated. In order to describe this method, we need to keep track of the remaining work for the tasks because it will no longer be proportional to the task's utilization. We let $\xi_{i,j}$ denote the remaining work to be done on job $T_{i,h}$ at the beginning of time slice $\sigma_j \in \Psi_{i,h}$. Thus,

$$\xi_{i,j} = e_i - \sum_{\sigma_k \in \Psi_{i,h} \wedge k < j} \ell'_{i,k}.$$

With implicit swapping, Step 1 of the algorithm above will initialize ℓ_{i,t_j} to $X_k \cdot (e_i - \xi_{i,j})/(d_{i,h} - t_j)$ instead of $X_k \cdot u_i$. An implicit swap makes the following adjustments between T_A and T_B.

- $\ell_{A,t_{j_1}} \leftarrow \ell_{A,t_{j_1}} + a$,
- $\ell_{B,t_{j_1}} \leftarrow \ell_{B,t_{j_1}} - a$,
- $\xi_{A,t_{j_1+1}} \leftarrow \xi_{A,t_{j_1+1}} - a$, and
- $\xi_{B,t_{j_1+1}} \leftarrow \xi_{B,t_{j_1+1}} + a$.

We let $\omega_{i,t}$ denote T_i's *remaining utilization* at time t – namely

$$\omega_{i,t} = \frac{e_i - \xi_{i,t}}{d_i - t},$$

where d_i is T_i's next deadline after t. Clearly, at all times t, we must ensure that $\omega_{i,t} \leq 1$ for all tasks T_i. If we also ensure the following rule is always satisfied for all time slices σ_j

$$\sum_{i=1}^{n} \omega_{i,t_j} \leq m,$$

then we can be sure that no deadline will be violated.

For example, in Figure 1 (assuming no execution times are altered in σ_1) we might increase $\ell_{4,8}$ and $\ell_{1,8}$ to 2, allowing these two tasks to executes to full length of σ_2. Also, $\ell_{3,8}$ must be 1.2 because T_3's deadline is 10. Because only 6 units of work can be completed during σ_2, we must reduce $\ell_{2,8}$ to 0.8 instead of 1.25. Thus, the reduced 0.45 units of work would be spread out through the time slices σ_3, σ_4, σ_5. At time $t = 10$, task T_2 would have 4.2 units of work remaining instead of 3.75 units of work.

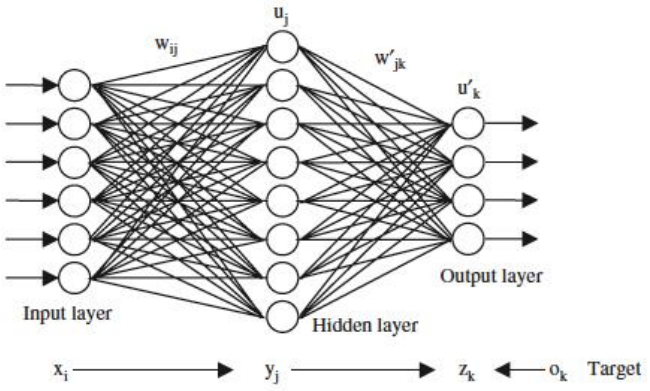


Fig. 2. An illustration of a Neural Network

Maintaining this condition is not as easy as it first seems. If $d_A < d_B$ when the swap occurs, then $\omega_{B,t_j+1} > u_B$ and $\omega_{A,t_j+1} < u_A$. At time d_A, when T_A generates its next job, ω_{A,d_A} will become u_A. At that point, it could be possible that the above rule is violated if ω_{B,d_A} is still larger than u_B.

Having developed these rules, our next step will be to implement NQ-Wrap and these rule sets to approximate a constant time near-optimal solution to minimize the number of preemptions and migrations, which is NP-complete in nature. Machine Learning techniques are capable of handling such rule sets efficiently. Our intent here is to estimate a constant time approximation of the rule sets whose application is NP-complete.

V. Machine Learning

As discussed above, we will be dealing with back-propagation neural networks (BPN_S), or multi-layer perceptrons[8]. These are one of the most well known tools and are widely used as pattern classifiers and function approximators[4]. Figure 2 gives an overview of a neural network of this type. As can be seen, there is a single hidden layer and there are no connections between neurons in the same layer in this particular case.

The back-propagation training algorithm is the one that is used in this type of neural network. There are several versions of this algorithm, but we consider the standard neural network model as described below. We will assume a network with an input layer of n_1 neurons, a hidden layer of n_2 neurons, and an output layer of n_3 neurons. The input, hidden and the output layers are x_i, y_j and z_k respectively. The weights of the connections that connect the first two layers and the thresholds of the second layer neurons are called w_{ij} and u_j, respectively. Similarly, w'_{jk} and u'_k, are the weights of the connections between the two latter layers and the thresholds of the third layer neurons.

The training algorithm is iterative, and employs the gradient algorithm to minimize a function that measures the difference between the network output (z_k^μ) and the desired one (o_k^μ). This algorithm has two phases, one forward and the other backward. The former calculates the difference between the network output and the desired one:

$$E(w_{ij},u_j,w'_{jk},u'_k) =$$

$$\frac{1}{2}\sum_{\mu=1}^{p}\sum_{q=1}^{n_3}\left[(o_q^\mu) - f\left(\sum_{j=1}^{n_2} w'_{jq} \cdot y_j^\mu - u'_q\right)\right]^2,$$

where $y_j^\mu = f\left(\sum_{i=1}^{n_1} w'_{ij} \cdot x_i^\mu - u'_j\right)$.

Here, p is the number of training examples.

This function called the cost, target, error or energy function measures how appropriate the weights of the connections are, and approaches zero when the network output approximates to the desired output. Once this function has been calculated, the backward phase follows. In this phase, by applying the gradient algorithm, the weights and thresholds are modified so that error is reduced. This process is iterated until the error is reduced to the desired amount. Finally, inductive learning algorithms generate a decision tree from a set of training examples.

For the training algorithm we have, the task set τ, optimal schedules deduced from the rule set discussed above, number of processors and the Sigmoid activation function[6]. For a particular *time slice*, we have to take care of the amount of execution time left for each task, the amount of time left to reach deadlines of each task, and the amount of execution time intended for each task in a time slice. In addition, we have the length of time slices as global parameters, irrespective of which time slice you are in.

For each time slice there will be one neural network per task. Also, there will be a neural network for the time slice. The task's neural network will each have 3 inputs and 3 outputs, one for remaining work, one for remaining time to deadline and one for work done during the time slice in consideration, each. The output nodes are replicated for feed-forward and feedback. In addition, the time slice has 3 additional inputs. These are flags dealing with the amount of execution time left on each processor, so as to switch between the schedules allocating a new execution time for the task sets. We set a *negative* and a *zero* flag for taking care of the same. For example, each time slice as discussed in Figure 1 will have $2 * 4 * 3 + 3 = 27$ input nodes and $6 * 4 = 24$ output nodes.

Finally, we use *back-propagation* for error feedback and the stopping criteria for the training network would be the error rates going up on the test sets versus the training sets.

VI. Future Work

Once this work is developed for periodic tasks with deadlines equal to periods, we plan to consider sporadic tasks and tasks with deadlines need not be constrained. We then plan to incorporate other more challenging changes.

We plan to make the schedule quantum based in the sense that each job executes for an integer amount of time within each time slice. This reduces the number of times that preemptions and migrations can occur. Of course, in this case task parameters must be integers.

Also, we plan to account for resource sharing or non-preemptive sections. All work to date has assumed an independent set of fully preemptable tasks. Abandoning either of those assumptions will introduce many challenges.

Finally, for the machine learning part, we might want to add the concept of *cross validation* as a future scope.

References

[1] Björn Andersson and Konstantinos Bletsas. Sporadic multiprocessor scheduling with few preemptions. In *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pages 243–252, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Dario Faggioli, Michael Trimarchi, and Fabio Checconi. An implementation of the earliest deadline first algorithm in linux. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1984–1989, New York, NY, USA, 2009. ACM.

[3] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pages 13–22, Washington, DC, USA, 2008. IEEE Computer Society.

[4] Richard P. Lippmann. An introduction to computing with neural nets. pages 36–54, 1988.

[5] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[6] Sridhar Narayan. The generalized sigmoid activation function: competitive supervised learning. *Inf. Sci.*, 99(1-2):69–82, 1997.

[7] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[8] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. pages 696–699, 1988.

Feasibility Test for Multi-Phase Parallel Real-Time Jobs

Vandy Berten, Sébastien Collette, Joël Goossens
Computer Science Department, U.L.B., Brussels, Belgium
Supported by the F.R.S.-FNRS and the Communauté Française de Belgique.
{*vandy.berten, sebastien.collette, joel.goossens*}*@ulb.ac.be*

***Abstract*—We consider the scheduling of real-time and parallel tasks on identical multiprocessor platforms (see [1], [2]). We deal with the case where each job may be executed on different processors *simultaneously*, i.e., we allow *job parallelism*. Jobs are composed of phases to be executed sequentially. We provide a *necessary* feasibility test and we believe that our techniques will yield a sufficient test.**

I. Introduction

In this research we study the feasibility problem, i.e., the existence of a feasible schedule which meets the timing requirements. This work can be seen as a generalization of the seminal paper on parallel and real-time jobs (see [3] for details). We study a task model which is more flexible than [3] in the sense that a task can have an arbitrary number of phases that have to be executed sequentially (see [4] for details) while [3] considered a single phase per task. In our model, each phase is characterized by a (maximal) degree of parallelism. E.g., the task could be composed of initialization, computing and finalization phases; solely the computing phase can be parallelized, others are sequential.

II. Model

In our model the platform is composed of m identical processors. We consider the scheduling of n frame-based and implicit-deadline real-time tasks $\tau_1, \tau_2, \ldots, \tau_n$, i.e., all tasks share the same period or deadline. In the following, D denotes the frame length, and as we manage each frame independently, we denote by $t = 0$ the beginning of each frame. Each task τ_i is composed of a sequence of q_i phases: $\tau_i =< \tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,q_i} >$.

Each phase $\tau_{i,k}$ is characterized by two parameters: $w_{i,k}$ denotes the required amount of work of $\tau_{i,k}$, and $P_{i,k}$ denotes the maximal degree of parallelism of $\tau_{i,k}$, i.e., the phase can scheduled on at most $P_{i,k}$ processors simultaneously and the speedup is assumed to be the number of assigned processors (i.e., fully parallelism is assumed). This model is presented in [4].

In this work we assume that the number of CPUs can vary inside a phase, and that each task could be assigned to a *fractional* number of processors at each time instant. Preemptions, migrations, and parallelism changes can occur at any (rational) time. Finally, we assume that the characteristics of the system are given by rational numbers ($w_{i,k}$, D, etc.). While these assumptions can seem theoretical, an exact feasibility test in this model gives a necessary condition for more realistic models, as a system which is infeasible even if we assign fractions of processors to each task, will certainly not be feasible in a more constrained setting.

III. Definitions and properties

A. Schedule and trajectory

Our main idea to derive feasibility tests is to apply our reasoning on *schedules* and *trajectory functions*:

Definition 1: A *schedule* $\{\varsigma_{i,k}(x)\}$ is a set of $\sum_{i=1}^{n} q_i$ functions giving for each phase $\tau_{i,k}$ and any time x the number of processors allocated to this phase. More formally, $\varsigma_{i,k}(x) \in [0, m]\ \forall i \in \{1, \ldots, n\}, \forall k \in \{1, \ldots, q_i\}$.

Notice that, since we consider identical multiprocessors and the fact that preemption/migration costs are neglected, without loss of generality we do not need to specify in Def. 1 on which processor[s] a phase is executed.

Definition 2: A schedule is said to be *valid* if (Eq. 1) it respects all the deadlines, (Eq. 2) respects the precedence constraints, (Eq. 3) does not use more resources than available, (Eq. 4) and tasks do not use more processors than what their parallelism allows:

$$\forall i \in \{1, \ldots, n\}, k \in \{1, \ldots, q_i\} : \int_0^D \varsigma_{i,k}(x)dx = w_{i,k} \tag{1}$$

$$\forall i \in \{1, \ldots, n\}, k \in \{1, \ldots, q_i - 1\}, \exists x \text{ such that } \forall x' \geq x : \varsigma_{i,k}(x') = 0 \wedge \forall x' \leq x : \varsigma_{i,k+1}(x') = 0 \tag{2}$$

$$\forall x \in [0, D] : \sum_{i=1}^{n} \sum_{k=1}^{q_i} \varsigma_{i,k}(x) \leq m \tag{3}$$

$$\forall i \in \{1, \ldots, n\}, k \in \{1, \ldots, q_i\}, x \in [0, D], \varsigma_{i,k}(x) \leq P_{i,k}. \tag{4}$$

Definition 3: Given a schedule, the *trajectory* $\sigma_{i,k}(x)$ of a phase $\tau_{i,k}$ is a continuous, monotonic function $x \to \sigma_{i,k}(x) : [0, D] \to [0, w_{i,k}]$, giving the total amount of work that the phase performed until time x. More formally: $\sigma_{i,k}(x) \stackrel{\text{def}}{=} \int_0^x \varsigma_{i,k}(t)\ dt$.

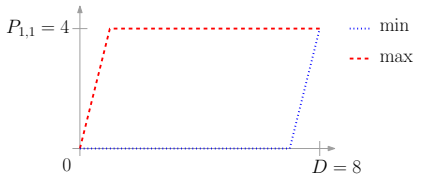


Figure 1. A single task composed of one phase ($\tau_1 =< \{4, 4\} >$), and its corresponding minimum and maximum trajectories.

Notice that[1]: $\sigma_{i,k}(0) = 0$, $\forall x \in (0, D] : \frac{d^-}{dx}\sigma_{i,k}(x) \leq P_{i,k}$ and $\varsigma_{i,k} = \frac{d^-}{dx}\sigma_{i,k}$.

Definition 4: A *valid trajectory* is a trajectory $\sigma_{i,k}$ such that there exists a valid schedule in which there is a function $\varsigma_{i,k}$ with $\varsigma_{i,k} = \frac{d^-}{dx}\sigma_{i,k}$.

Notice that if a trajectory is valid, then $\sigma_{i,k}(D) = w_{i,k}$.

Of course, different valid schedules for the same system imply the existence of different valid trajectories. Fig. 1 shows the example of a system composed of a single task, which is itself composed of a single phase: $\tau_1 =< \{4, 4\} >$ (4 units of work to be run on at most 4 processors). We would like to run this system on 4 processors, with a deadline $D = 8$. We identified the minimum and maximum trajectories for this phase (see Fig. 1).

The minimum valid trajectory is such that the quantity of work $\sigma_{1,1}(x)$ is minimized over each time instant. As the function is monotonic and has to reach $\sigma_{1,1}(D) = w_{1,1}$, the minimum valid trajectory is uniquely defined. Another constraint is that the maximum derivative of the function is $P_{1,1}$. The maximum valid trajectory is defined symmetrically.

B. Bounds

Definition 5: $UB_{i,k}(x)$ (resp. $LB_{i,k}(x)$) denotes, for any $x \in [0, D]$, an upper bound (resp. a lower bound) on the amount of work which can be processed in the interval $[0, x]$ for the phase $\tau_{i,k}$.

A very trivial value for those two bounds is $UB_{i,k}(x) \leftarrow w_{i,k}$, and $LB_{i,k}(x) \leftarrow 0$. For one phase, as explained above and illustrated on Fig. 1, we can provide better bounds:

$$LB_{i,k}(x) \leftarrow \begin{cases} 0 & x < D - \frac{w_{i,k}}{P_{i,k}} \\ P_{i,k} \cdot (x - D) + w_{i,k} & \text{otherwise.} \end{cases} \quad (5)$$

$$UB_{i,k}(x) \leftarrow \begin{cases} x \cdot P_{i,k} & x < \frac{w_{i,k}}{P_{i,k}} \\ w_{i,k} & \text{otherwise.} \end{cases} \quad (6)$$

Lemma 1: Any valid *trajectory* $\sigma_{i,k}(\cdot)$ is such that $LB_{i,k}(x) \leq \sigma_{i,k}(x) \leq UB_{i,k}(x)$ $\forall x \in [0, D]$, where $LB_{i,k}$ and $UB_{i,k}$ are defined by Eq. (5) and (6).

[1]In this document $\frac{d^-}{dx}$ denotes the left-sided derivative. Remark that the derivative is not defined at instants where the number of processors allocated to a task changes. We avoid this issue by using the left-sided derivative, which is always defined except for $x = 0$.

Proof: First we prove that a valid trajectory cannot be less than the lower bound. By contradiction, imagine there is a valid trajectory $\sigma_{i,k}(\cdot)$ such that, for some $x \in [0, D]$, $LB_{i,k}(x) > \sigma_{i,k}(x)$. Notice that this implies that $x > D - \frac{w_{i,k}}{P_{i,k}}$, because when $x \leq D - \frac{w_{i,k}}{P_{i,k}}$, $0 \leq \sigma_{i,k}(x)$ and $LB_{i,k}(x) = 0$ which means that $\sigma_{i,k}(x) \geq LB_{i,k}(x)$.

As the derivative is upper bounded by $P_{i,k}$,

$$\begin{aligned} \sigma_{i,k}(D) &\leq \sigma_{i,k}(x) + P_{i,k}(D - x) \\ &< LB_{i,k}(x) + P_{i,k}(D - x) = w_{i,k} \end{aligned}$$

and thus $\sigma_{i,k}(\cdot)$ is not valid, a contradiction.

It remains to prove that a valid trajectory cannot be more than the upper bound, also by contradiction: imagine that for some $x \in [0, D]$, $UB_{i,k}(x) < \sigma_{i,k}(x)$. This implies that $x < \frac{w_{i,k}}{P_{i,k}}$, as $\sigma_{i,k}(x) \leq w_{i,k}$. Thus:

$$\begin{aligned} \sigma_{i,k}(0) &\geq \sigma_{i,k}(x) - x \cdot P_{i,k} \\ &> UB_{i,k}(x) - x \cdot P_{i,k} = 0 \end{aligned}$$

and $\sigma_{i,k}(\cdot)$ is not valid, a contradiction. ■

IV. One task, one phase

We will now consider a very simple system with only one task, containing one phase, $\tau_{1,1}$. The feasibility of such a system is of course trivial, but we will formalize several notions that we will then extend to more general systems.

In fact, if we have a single phase, as there are no other constraints, the lower and bounds we gave in Eq. (5) and (6) of that phase completely characterize the set of valid schedules in the following sense: any trajectory, contained inbetween the lower and the upper bound, is valid.

Lemma 2: Given a feasible system composed of a single phase, and given a point $p = (p_1, p_2)$ such that $LB_{1,1}(p_1) \leq p_2 \leq UB_{1,1}(p_1)$, there exists a valid trajectory such that $\sigma_{1,1}(p_1) = p_2$.

Proof: First notice that in the case of a single phase, $LB_{1,1}(\cdot)$ and $UB_{1,1}(\cdot)$ given by Eq. (5) and (6) are valid trajectories: these are monotonic functions, such that $LB_{1,1}(0) = 0 = UB_{1,1}(0)$ and $LB_{1,1}(D) = w_{1,1} = UB_{1,1}(D)$, and whose derivative is at most $P_{1,1}$. As long as $P_{1,1} \leq m$, the derivative of these functions is a valid schedule for one phase.

Let t_1 be the smallest value such that $UB_{1,1}(t_1) = p_2$ and t_2 be the greatest value such that $LB_{1,1}(t_2) = p_2$. We define a new function $f(\cdot)$ as follows:

$$f(x) = \begin{cases} UB_{1,1}(x) & \text{if } x < t_1 \\ LB_{1,1}(x) & \text{if } x > t_2 \\ p_2 & \text{otherwise.} \end{cases}$$

We claim that $f(p_1) = p_2$ and that $f(\cdot)$ is a valid trajectory. Indeed, $t_1 \leq p_1 \leq t_2$, by definition of t_1 and t_2, therefore $f(p_1) = p_2$, as claimed. It remains to show

that $f(\cdot)$ is a valid trajectory: $f(\cdot)$ is continuous, monotonic, $f(0) = 0$ and $f(D) = w_{1,1}$, and

$$\frac{d^-}{dx} f(x) \leq \max \left\{ \frac{d^-}{dx} UB_{1,1}(x), \frac{d^-}{dx} LB_{1,1}(x) \right\}$$

from which we deduce that the corresponding schedule is valid. ■

It remains to show a full characterization of valid trajectories:

Theorem 1: Given a system composed of a single phase, a trajectory $\sigma_{1,1}(\cdot)$ is valid *if and only if*:

$$\forall x \in [0, D] : LB_{1,1}(x) \leq \sigma_{1,1}(x) \leq UB_{1,1}(x)$$

where $LB_{1,1}$ and $UB_{1,1}$ are defined in Eq. (5) and (6).

Proof: The theorem is a direct consequence of Lemmas 1 and 2: any trajectory for which $\forall x : LB_{1,1}(x) \leq \sigma_{1,1}(x) \leq UB_{1,1}(x)$ is valid (Lem. 2), and any trajectory for which there exists an x such that either $LB_{1,1}(x) > \sigma_{1,1}(x)$ or $UB_{1,1}(x) < \sigma_{1,1}(x)$ is not valid (Lem. 1). ■

Corollary 1: Given a system composed of a single phase, it is feasible if and only if

$$\forall x \in [0, D] : LB_{1,1}(x) \leq UB_{1,1}(x)$$

Proof: This is easily deduced form Thm. 1: if $LB_{1,1}(x) \leq UB_{1,1}(x)$, there exists a trajectory $LB_{1,1}(x) \leq \sigma_{1,1}(x) \leq UB_{1,1}(x)$, which is valid. If not, by Lem. 1, no valid trajectory exists. ■

V. Several phases, several tasks

In the general case, Thm. 1 is not sufficient anymore: our bounds are not tight and need to be improved. We will first define two operators that can be applied to our upper and lower bounds, to deduce tighter bounds.

Consider that we have a lower bound $LB_{i,k}(x)$ on the minimum trajectory of some phase $\tau_{i,k}$, and that we know (we will see later how) that the minimum reaches at least a value v at time u. Then, we can deduce a new lower bound, $LB'_{i,k}(\cdot)$, such that $LB'_{i,k}(u) \geq v$, as illustrated on Fig. 2. However, we have more information: as we know that a trajectory is monotonic and that its derivative is bounded by the maximum number of processors on which the phase can be scheduled ($P_{i,k}$), we deduce that updating the lower bound at coordinate x implies that

- $\forall x < u, LB'_{i,k}(x) \geq P_{i,k}(x - u) + v$, because the derivative is at most $P_{i,k}$.
- $\forall x > u, LB'_{i,k}(x) \geq v$, because the function is monotonic;

From this we can define the new lower bound, combining an LB function with a point (u, v). Therefore, we define an operator $\bigwedge$ receiving a lower bound trajectory function and a point, and resulting in a new lower bound trajectory function:

$$\bigwedge\{LB_{i,k}, (u, v)\}(x) \stackrel{\text{def}}{=} \begin{cases} \max\{P_{i,k} \cdot (x - u) + v, LB_{i,k}(x)\} & \\ & \text{if } x < u \\ \max\{v, LB_{i,k}(x)\} & \text{otherwise.} \end{cases}$$

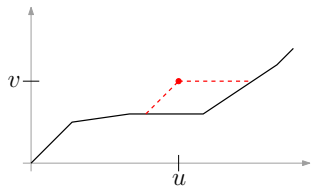


Figure 2. Updating the minimum of a lower bound on a trajectory at instant u might imply to update an interval of values.

Similarily, we define an operator $\bigvee$:

$$\bigvee\{UB_{i,k}, (u, v)\}(x) \stackrel{\text{def}}{=} \begin{cases} \min\{v, UB_{i,k}(x)\} & \text{if } x < u \\ \min\{P_{i,k} \cdot (x - u) + v, UB_{i,k}(u)\} & \\ & \text{otherwise.} \end{cases}$$

Even though a trajectory is a continuous function, it is piecewise linear, and updating lower or upper bounds implies to add at most two segments to the function. We can thus, using an appropriate data structure, maintain lower and upper bounds such that the space required is linear in the number of updates.

A. Phase precedence

As defined by our model, each task is composed of a certain number of phases, which have to be executed *sequentially*. This implies that the first phase must be finished before any subsequent phase of the same task starts, and every phase has to wait for the completion of previous phases before starting. This can be translated into the following facts:

Fact 1: Let u be the last instant where $LB_{i,k}(u) = 0$. Then, u is also the latest time at which every phase $\tau_{i,k'}$, with $k' < k$ has to finish, and $\sigma_{i,k'}(u) = w_{i,k'}$.

Indeed, u is the latest time at which it is possible that $\tau_{i,k}$ has not started yet, therefore, it is also the latest time at which every phase $\tau_{i,k'}$, with $k' < k$ has to finish.

Fact 2: Let u be the earliest instant where $UB_{i,k}(u) = w_{i,k}$. Then, every phase $\tau_{i,k'}$ with $k' > k$ cannot have started yet, and $\sigma_{i,k'}(u) = 0$.

This follows from the fact that the earliest time at which the task $\tau_{i,k}$ could possibly have finished its work is u. These two facts can be translated in terms of a rule allowing to update the bounds:

Rule 1: (Precedence Rule)

$\forall i \in \{1, \ldots, n\}, k \in \{q_i - 1, q_i - 2, \ldots, 1\} : LB_{i,k} \leftarrow \bigwedge\{LB_{i,k}, (u, w_{i,k})\}$, where $u = \max\{x \mid LB_{i,k+1}(x) = 0\}$.

$\forall i \in \{1, \ldots, n\}, k \in \{2, \ldots, q_i\} : UB_{i,k} \leftarrow \bigvee\{UB_{i,k}, (u, 0)\}$, where $u = \min\{x \mid UB_{i,k-1}(x) = w_{i,k-1}\}$.

Notice that the rule has to be applied as in the definition with k going from $q_i - 1$ down to 1 for the first part, and from 2 up to q_i. If not, the rule needs to be applied several times until stabilization.

If our system is composed of a single task with multiple phases, the precedence rule gives necessary and sufficient

conditions on the schedulability of the system:

Theorem 2: Given a system composed of a single task τ_1 with q_1 phases, a trajectory $\sigma_{1,k}(\cdot)$ is valid *if and only if*:

$$\forall x \in [0, D], 1 \leq k \leq q_i : LB_{1,k}(x) \leq \sigma_{1,k}(x) \leq UB_{1,k}(x)$$

where $LB_{1,k}$ and $UB_{1,k}$ are defined according to Eq. (5) and (6) and Rule 1.

Proof: A system with one task is feasible if and only if all its phases can be achieved sequentially. After applying Rule 1, $LB_{1,k}(\cdot)$ and $UB_{1,k}(\cdot)$ are valid trajectories, as long as $\forall x : LB_{1,k}(x) \leq UB_{1,k}(x)$. Indeed, these are monotonic functions, such that $LB_{1,k}(0) = 0 = UB_{1,k}(0)$ and $LB_{1,k}(D) = w_{1,k} = UB_{1,k}(D)$, and whose derivative is at most $P_{1,k}$, and such that each phase does not start as long as all other phases $\tau_{i,k'}$ with $k' < k$ could not have finished their work. We deduce from this that the condition is sufficient.

To prove that it is also necessary, we simply notice that a valid trajectory for which there exists an x such that either $LB_{1,k}(x) > \sigma_{1,k}(x)$ or $UB_{1,k}(x) < \sigma_{1,k}(x)$ contradicts Fact 1 or 2, which are necessary conditions. ■

Corollary 2: Given a system composed of a single task, it is feasible if and only if $\forall x \in [0, D]$, $1 \leq k \leq q_i$: $LB_{1,k}(x) \leq UB_{1,k}(x)$

B. Task interactions

From the upper and lower bounds on a phase, we can also deduce the minimum and the maximum quantity of work that has to be achieved by a phase over some interval. For all x and x', such that $0 \leq x \leq x' \leq D$, the *minimum of work to be done* by phase $\tau_{i,k}$ over the interval $[x, x']$ is

$$\underset{i,k}{minwork}(x, x') \stackrel{\text{def}}{=} \begin{cases} LB_{i,k}(x') - UB_{i,k}(x) \\ \qquad \text{if } LB_{i,k}(x') - UB_{i,k}(x) \geq 0, \\ 0 \qquad\qquad\qquad \text{otherwise.} \end{cases}$$

while the maximum is $\underset{i,k}{maxwork}(x, x') \stackrel{\text{def}}{=} UB_{i,k}(x') - LB_{i,k}(x)$.

Fact 3: If a trajectory does not perform at least $minwork_{i,k}(x, x')$ and at most $maxwork_{i,k}(x, x')$ units of work over the interval $[x, x']$, the trajectory is not valid.

Therefore, $\forall x \leq x', \forall i, i', k, k'$:

$$\underset{i,k}{maxwork}(x, x') \leq m(x' - x) - \sum_{i' \neq i, k'} \underset{i',k'}{minwork}(x, x') \ .$$

And thus, if we know that there is some value v such that $maxwork_{i,k}(x, x') \leq v$, we get that $UB_{i,k}(x') \leq UB_{i,k}(x)+v$, and $LB_{i,k}(x) \geq LB_{i,k}(x')-v$. These conditions translate into the following rule:

Rule 2: (Interaction Rule) $\forall x \leq x', \forall i, i', k, k'$:

$$LB_{i,k} \leftarrow \bigwedge\{LB_{i,k}, (x, LB_{i,k}(x') - v)\},$$

$$UB_{i,k} \leftarrow \bigvee\{UB_{i,k}, (x', UB_{i,k}(x) + v)\}$$

where $v = m(x' - x) - \sum_{i' \neq i} minwork_{i',k'}(x, x')$.

VI. Feasibility

Theorem 3: If after applying rules 1 and 2 a certain number of times on the bounds given by Eq. (5) and (6), it occurs that it is not possible, for a task $\tau_{i,k}$ to find any valid trajectory in the area defined by $UB_{i,k}$ and $LB_{i,k}$, *then* the system is not feasible. In other words: $\exists x, i, k : UB_{i,k}(x) < LB_{i,k}(x) \Rightarrow$ the system is not feasible.

Proof: Applying rules 1 and 2 is equivalent to giving bounds where Fact 1, 2 and 3 are verified. As these facts are necessary conditions for a trajectory to be valid, the theorem is a necessary condition for schedulability. ■

VII. Ongoing Work

Let $UB^{\infty}_{i,k}(x)$ and $LB^{\infty}_{i,k}(x)$ be the bounds we obtain by applying Rules 1 and 2 until stabilization, meaning that applying one of the rules on the bounds will not change them anymore[2]. Let $UB(x) \stackrel{\text{def}}{=} \sum_{i,k} UB^{\infty}_{i,k}(x)$ and $LB(x) \stackrel{\text{def}}{=} \sum_{i,k} LB^{\infty}_{i,k}(x)$. Let $UB^*(x)$ be the maximal monotonic function such that

$$\forall x : UB^*(x) \leq UB(x), \text{ and } \forall x > 0 : \frac{d^-}{dx} UB^*(x) \leq m$$

and symmetrically $LB^*(x)$ be the minimal monotonic function such that

$$\forall x : LB^*(x) \geq LB(x), \text{ and } \forall x > 0 : \frac{d^-}{dx} UB^*(x) \leq m.$$

Conjecture 1:

$$\forall x, i, k : UB^*(x) \geq LB^*(x) \Leftrightarrow \text{the system is feasible}$$

We conjecture that, after applying our rules, it is possible to decide if a system is feasible by simply comparing two functions. This would provide an *exact* feasibility test.

Acknowledgment: We would like to thank Prof. S. Baruah, for posing the problem and for interesting discussions.

References

[1] T. P. Baker and S. K. Baruah, *Handbook of Real-Time and Embedded Systems*. Chapman and Hall, 2007, ch. Schedulability Analysis of Multiprocessor Sporadic Task Systems, pp. 3–1 – 3–15.

[2] T. P. Baker, "An analysis of EDF scheduling on a multiprocessor," *IEEE Trans. on Parallel and Distributed Systems*, vol. 15, no. 8, pp. 760–768, 2005.

[3] S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Information Processing Letters*, vol. 106, no. 5, pp. 180–187, May 2008. [Online]. Available: http://arxiv.org/abs/0805.3237

[4] J. Edmonds, "Scheduling in the dark," *Theor. Comput. Sci.*, vol. 235, no. 1, pp. 109–141, 2000.

[2]Of course, we have to prove that we can reach the stable state in finite time, implying that (1) the process converges and (2) that the order in which the rules are applied does not change the result.

Virtual Timers in Hierarchical Real-time Systems

Martijn M.H.P. van den Heuvel, Mike Holenderski, Wim Cools, Reinder J. Bril and Johan J. Lukkien
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands

***Abstract*—Hierarchical scheduling frameworks (HSFs) provide means for composing complex real-time systems from well-defined subsystems. This paper describes an approach to provide hierarchically scheduled real-time applications with virtual event timers, motivated by the need for integrating priority processing applications in an HSF. Specifically, the paper proposes a technique to minimize the overhead of event handling in HSFs and outlines a simple implementation.**

I. Introduction

The increasing complexity of real-time systems demands for a decoupling between (*i*) development and analysis of individual applications and (*ii*) integration of applications on a shared platform, including the analysis at the system level. Hierarchical scheduling frameworks (HSFs) have been extensively investigated as a paradigm to facilitate this decoupling, see for example [1]. In this paper we consider a two level HSF, where a system is composed of a set of independent *applications*, each of which is composed of a set of *tasks*. A *server* is allocated to each application. A global scheduler is used to determine which server should be allocated the processor at any given time. A local scheduler determines which of the chosen application's tasks should actually execute.

Multimedia applications define a well-studied class of real-time applications. To enable cost-effective media processing in software, scalable video algorithms (SVAs) have been developed that allow trading quality for resource needs. The principle of priority processing provides optimal real-time performance for scalable video algorithms on programmable platforms even with limited system resources [2]. According to this principle, SVAs provide their output strictly periodically and processing of images follows a priority order. Hence, important image parts are processed first, followed by less important parts in a decreasing order of importance. After creation of an initial output by a basic function, processing can be *preliminary terminated* at an arbitrary moment in time, yielding the best output for given resources, see Figure 1.

To distribute the available resources, i.e. CPU-time, among independent priority processing algorithms, an application specific strategy has been developed [3]. This strategy is implemented in a *decision scheduler* and aims at maximizing the total relative progress of the SVAs. The relative progress of an algorithm is defined in terms of the fraction of the performed work relative to the consumed budget and the total amount of work to be done in a video frame.

As a leading example, we consider a priority processing application, composed of multiple independent SVAs and a decision scheduler, which divides the available virtual processor resources among the SVAs, as described in [4]. They assume that the application has the full processor at its disposal.

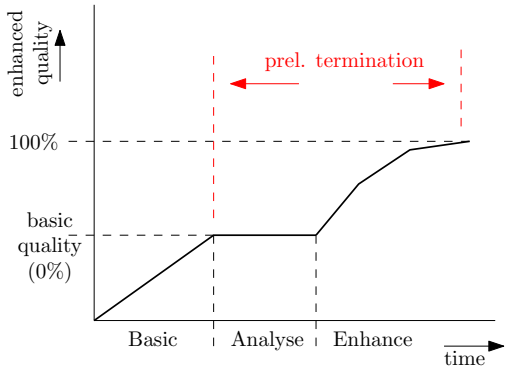


Fig. 1. Priority processing, as a function of time consumption versus output quality, can be divided in three time-frames: 1) produce a basic output at the lowest quality level; 2) Identify the most important image content; 3) Enhance the quality of the output by processing the most important picture parts first.

A. Problem Description

In this paper we consider the scenario in which a priority processing application is provided a virtual share of the available processor resources, by assigning the decision scheduler and the SVAs a single virtual processor.

The decision scheduler implements the control strategy and divides the available processor time within the application budget into fixed-sized quanta termed *time-slots* of the size Δt_s. The control strategy selects the SVA to execute next upon completion of a time-slot, i.e. synchronous with virtual time. Activation of the decision scheduler is triggered by the depletion of a time-slot. Hence, the application requires *virtual timers* to trigger timed events *relative to the consumed budget*, to activate the decision scheduler for monitoring the progress of the SVAs.

B. Contributions

Given the need for virtual timer events on shared virtual platforms, we outline a low-overhead implementation of virtual timers, targeted at embedded systems. Additionally, the outlined solution aims at minimizing the overhead of handling events of inactive applications in HSFs.

C. Outline

The remainder of this paper is as follows. Section II describes the related work. Section III describes the virtual

platform model used as a reference for our implementation directions. Section IV describes an approach to realize virtual timed events. Finally, Section V concludes the paper.

II. Related work

The notion of a virtual timer within an application already exists for POSIX-compliant operating systems [5]. Each process running on such a platform has the availability of a virtual timer that counts processor time used by that process. When the virtual timer expires, a signal is sent to the process. Upon expiration of a timer, the corresponding signal is queued for the corresponding process, whereas arrival of a signal depends on the granularity of the kernel clock. Signals, as described in the POSIX standard, are a form of inter-process communication. Processes are the primitive units for allocation of system resources. Each process has its own address space and one thread of control. When considering our priority processing application, it is natural to map the application on a single process. The decision scheduler and the SVAs are each mapped on its own thread contained in the process, whereas a thread is used as a scheduling unit. We require signalling of the decision scheduler's thread upon expiration of the virtual timer, instead of the main process. Although the concept is similar, we require a more general notion of virtual timers.

Partitioning the system to independent subsystems, which are each provided with a virtual platform, is currently researched in two slightly different directions. On the one hand, directions go towards hierarchical scheduling schemes. Hierarchical real-time system development is based on sound analysis and well-defined application interfaces [1]. On the other hand, virtual real-time operating systems (RTOSs) are investigated to obtain a strong partitioning of the system [6], [7]. Main challenges with respect to satisfying real-time constraints in virtualizing a RTOS are (*I*) increasing responsiveness with respect to hardware interrupts and (*II*) synchronization of virtual machine related timer events. Both issues require a low level virtual machine monitor to manage interrupts and events messages [7]. Issue (*I*) includes the problem that an interrupt can be generated for a particular virtual machine which is not assigned to the processor at that moment in time. The virtual RTOS is not allowed to disable interrupts, which entails additional demands on the virtual machine's monitoring layer. The virtual machine monitor has to queue all interrupts for an inactive RTOS and block the interrupts for an RTOS when it requests to temporarily disable interrupts. Issue (*II*) relates to managing event queues. When a virtual RTOS is active, it can handle all timer events directly. All queued timer events for a particular inactive virtual RTOS, as addressed by issue (*I*), must be synchronized with the local event queue upon activation of the virtual machine. Support for virtual timers, as required by our priority processing application, is lacking in the description in [6], [7].

[8] presents a novel design for managing timed event queues, RELTEQ, applicable for embedded operating systems demanding low memory and processor overhead. Reservation based real-time systems provide applications with the facility to request their remaining budget within the current replenishment period. Reservation based kernels rely on mechanisms for *admission control*, *scheduling*, *monitoring* and *enforcement* [9]. Note that virtual timed events are different from monitoring the consumed budget within an application. Although RELTEQ can be exploited to support budget monitoring, it does not support the generalized concept of virtual timed events. Enforcement of budgets requires expiration of timers upon depletion of the budget. [10] implemented enforcement timers by setting a single timer on activation of a server, indicating the depletion of the server's budget. The enforcement timer is set to the minimum value of the remaining budgets of all levels in the hierarchical resource chain, e.g. levels in an HSF. Every budget has its own replenishment timer. Finally, all timers are added to a single global event queue.

[11] keeps track of budget depletion by using separate event queues for each server in the HSF by means of absolute times. On activation of a server, an event indicating the depletion of the budget, i.e. the current time plus the remaining budget, is added to the server event queue. On preemption of a server, the remaining budget is updated according to the time passed since the last server release and the budget depletion event is removed from the server event queue. When the server's budget depletion event expires, the server is removed from the server ready queue, i.e. it will not be rescheduled until the replenishment of its budget.

In this paper we show how to extend the RELTEQ [8] approach to manage virtual timed event queues.

III. Virtual Platform Model

Given an HSF mapped on a single processor, we consider a priority processing application, attached to a server within the HSF. For simplicity, we assume an *idling periodic server* [12], however the proposed approach is expected to be easily adaptable to other server types. We say that tasks assigned to a server consume processor time *relative to the server's budget* to signify that the consumed processor time is accounted to (and subtracted from) that budget.

Given the priority processing application, the decision scheduler task is assigned the highest priority, such that upon activation it can immediately preempt the SVAs. The SVAs are each mapped on a strictly periodic task. All SVAs are synchronous with the same period, P_f, i.e. each period the SVAs start with a new video frame and at the end of a period the processing is terminated. The SVAs are not blocked by their input and output and share no resources except the processor. All tasks comprising the priority processing application are assigned to the same server.

A server has a replenishment period, P_b, and a budget, Q_b. Activation of the decision scheduler, i.e. a virtually timed event, is triggered after consumption of a time-slot, Δt_s, relative to the budget Q_b. For convenience we assume that P_f is a multiple of P_b and has the same phasing. For example, the video frame rate $P_f = 20ms$, the application is provided with a budget $Q_b = 5.5ms$ every period $P_b = 10ms$, and $\Delta t_s = 1ms$. This scenario is sketched in Figure 2.

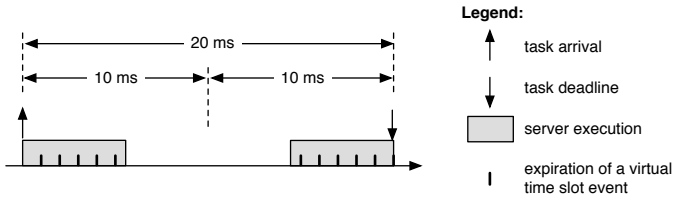


Fig. 2. Example of budget replenishments and virtual events, with $P_f = 20ms$, $P_b = 10ms$, $Q_b = 5.5ms$, and $\Delta t_s = 1ms$.

IV. Proposed Approach

We start this section by summarizing the RELTEQ [8] approach to multiplexing timed events on a single hardware timer. Then we describe how RELTEQ can be extended to support hierarchical scheduling. Finally, we outline an efficient RELTEQ implementation of virtual timers.

A. Basic RELTEQ timer management

RELTEQ stores the arrival times of events relative to each other, by expressing the arrival time of an event relative to the arrival time of the previous event. The arrival time of the head event is relative to the current time, as shown in Figure 3. While RELTEQ is not restricted to any specific hardware timer, in this paper we assume a periodic timer. At every tick of the periodic timer the time of the head event in the queue is decremented.

Two operations can be performed on an event queue: new events can be inserted and the head event can be popped. When a new event e_i with absolute time t_i is inserted, the event queue has to be traversed, accumulating the relative times of the events until a later event e_j is found, with $t_i < t_j$, where t_i and t_j are both absolute times. When such an event is found, then *(i)* e_i is inserted before e_j, *(ii)* its time is set relative to the previous event, and *(iii)* the arrival time of e_j is set relative to e_i. If no later event was found, then e_i is appended at the end of the queue, and its time is set relative to the previous event.

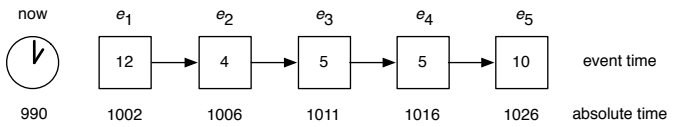


Fig. 3. Example of the RELTEQ event queue.

The first event: The arrival time of the first event is expressed in *absolute* time. To prevent the first event from overflowing, RELTEQ inserts *dummy events* at times when the absolute time would overflow, as shown in Figure 4.

The time will overflow once in 2^n ticks (assuming an n-bit time representation), requiring to insert one dummy event every 2^n ticks. Since the number of proper events within that time interval is likely to be high, the overhead of using dummy events to handle absolute time overflows is small.

[8] also describes how to use dummy events to provide unbounded interarrival times between events.

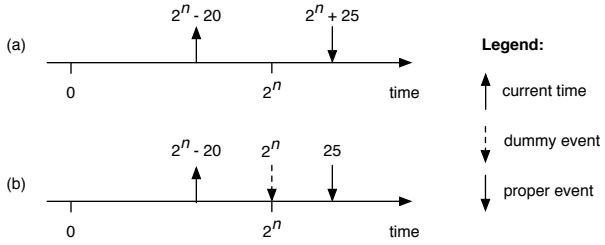


Fig. 4. Example of (a) overflowing absolute time of the first event (b) RELTEQ inserting a dummy event to handle the overflow.

B. Extending RELTEQ with hierarchical scheduling

The original description of RELTEQ [8] revolved around a periodic hardware timer driving a single event queue. To support hierarchical scheduling, we add an additional *server queue* for *each* server, to keep track of the events local to the server. At any time at most one server can be active; all other servers are inactive. The additional server queues make sure that the events local to inactive servers do not interfere with the currently active server.

In this new configuration the hardware timer drives two event queues:

1) the *system queue*, keeping track of events such as replenishment of periodic servers,
2) the *server queue* of the *active* server, keeping track of events such as task deadlines or the arrival of periodic tasks.

At every tick of the periodic timer the heads of both queues are decremented. The inactive server queues are left untouched.

When the active server is switched out (e.g. a higher priority server is resumed, or the active server gets depleted) then the active server queue is replaced by the queue belonging to the new active server. As a result, the queue of the switched out server will be "paused", and the queue of the switched in server will be "resumed".

To keep track of the time which has passed since the last server switch, we introduce one additional *stopwatch queue*. Initially it contains a single "dummy" event with time 0. At every tick of the periodic timer the head of the stopwatch queue is decremented. Time overflows are handled by setting the overflowing event to -2^n and inserting a new "dummy" event at the head of the queue with time equal to the overflow.

When the active server is switched out, the head event in the server queues of all inactive servers is decremented with the sum of all event times in the stopwatch queue, and the stop watch queue is reset to a single "dummy" event with time 0. Time overflows in the server queues are handled by inserting dummy events at the head of the queue, similar to handling overflows of the stopwatch queue.

When an inactive server is switched in, all leading events in its server queue are handled, until the head points to an event with a positive *absolute* event time. The absolute event times are computed in the same way as in the original RELTEQ,

by accumulating the relative times of subsequent events in the queue.

When the server budget is depleted an event must be triggered, to guarantee that a server does not exceed its budget. We could resolve the budget depletion events in a way similar to [11]. Because their approach requires to remove the budget depletion event from the server queue every time the server is switched out and to insert it back when the server is switched in, we opt for an alternative approach.

C. Extending RELTEQ with virtual timers

In Section I we have identified the need for "time-slot" events, which expire at times relative to the consumption of the server budget. In this section we present a general approach for handling both budget depletion and time-slot events and introduce the notion of *virtual timers*. Our approach avoids removing virtual events upon server switching and is therefore more efficient than that of [11].

We can implement virtual timers by adding a *virtual server queue* for *each* server. In this new configuration, at every tick of the periodic timer the heads of all four queues are decremented: system queue, active server queue, stopwatch queue, and active virtual server queue.

Similarly to the server queues introduced earlier, when a server is switched out, the active virtual server queue is paused and the switched in virtual server queue is resumed. The difference is that the stopwatch time is not subtracted from the head of the virtual server queue, since during the inactive period a server does not consume any of its budget.

An example of the proposed RELTEQ extension with hierarchical scheduling and virtual timers is shown in Figure 5.

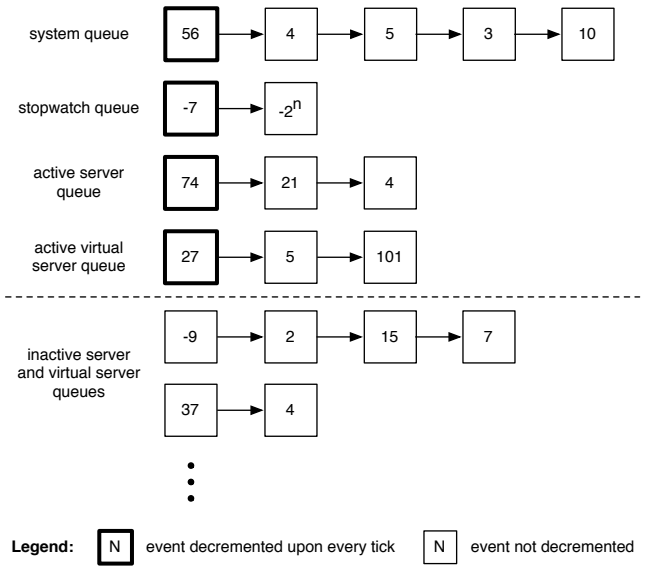


Fig. 5. Example of RELTEQ based implementation of reservations.

V. Conclusion

This paper generalizes the concept of a virtual timer to hierarchical real-time systems. Specifically, the paper proposes a technique to minimize the overhead of event handling in hierarchical scheduling frameworks that may be used in compositional design and analysis of complex real-time systems. In such systems, several applications execute on a shared processor where each application is given a virtual share of the processor and is responsible for local scheduling of tasks within itself. We outlined an implementation of hierarchical scheduling and virtual timers based on the RELTEQ approach to multiplexing timed events on a single hardware timer. The proposed implementation aims at minimizing the overhead of handling events belonging to inactive servers. In the future we would like to further investigate trade-offs between different design and implementation alternatives of HSFs with virtual timers in RELTEQ.

Our current research on providing temporal isolation between applications in real-time systems focusses on two-level HSFs. In the future work we would like to extend the proposed approach to multi-level hierarchical scheduling, where the hardware timer is driving a server queue for each server in the hierarchy of currently active servers.

References

[1] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proc. 24th IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2003, pp. 2–13.

[2] C. Hentschel and S. Schiemenz, "Priority-processing for optimized real-time performance with limited processing resources," in *Proc. 26th IEEE Int. Conference on Consumer Electronics (ICCE). Digest of Technical Papers.*, Jan. 2008.

[3] S. Schiemenz, "Echtzeitsteuerung von skalierbaren Priority-Processing Algorithmen," in *Tagungsband ITG Fachtagung - Elektronische Medien*, March 2009, pp. 108 – 113.

[4] M. van den Heuvel, R. J. Bril, S. Schiemenz, and C. Hentschel, "Dynamic resource allocation for real-time priority processing applications," in *Accepted for 28th IEEE Int. Conference on Consumer Electronics (ICCE). Digest of Technical Papers.*, Jan. 2010.

[5] GNU-Project. (2009, Sep.) Setting an alarm - the gnu c library. [Online]. Available: http://www.gnu.org/s/libc/manual/html_node/Setting-an-Alarm.html

[6] S. Yoo, M. Park, and C. Yoo, "A step to support real-time in virtual machine," in *Proc. 6th IEEE Consumer Communications and Networking Conference (CCNC)*, Jan. 2009, pp. 1–7.

[7] D. Kim, Y.-H. Lee, and M. Younis, "Spirit-μkernel for strongly partitioned real-time systems," in *Proc. 7th Int. Conference on Real-Time Computing Systems and Applications.*, 2000, pp. 73–80.

[8] M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Multiplexing real-time timed events," in *Proc. 14th IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA)*, July 2009.

[9] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *Proc. SPIE, Vol. 3310, Conference on Multimedia Computing and Networking (CMCN)*, January 1998, pp. 150–164.

[10] S. Saewong and R. Rajkumar, "Hierarchical reservation support in resource kernels," 2001. [Online]. Available: http://www.cs.cmu.edu/afs/cs/project/rtml-2/Papers/hrsv.ps.gz

[11] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards hierarchical scheduling on top of VxWorks," in *Proc. 4th Int. Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, July 2008, pp. 63–72.

[12] R. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Proc. 26th IEEE Int. Real-Time Systems Symposium (RTSS)*, Dec. 2005, pp. 389–398.

Analysis of Latest Defer Time for Fixed-Priority Scheduling Algorithm with Dual Priority

Chiahsun (Alex) Ho and Shelby Funk
Computer Science Department
University of Georgia
Athens, GA, USA
{*ho,shelby*}*@cs.uga.edu*

Abstract

We present an online adjustment method to discover the latest defer time for each job using Fixed Priority scheduling algorithm. By deferring the preemption of lower priority jobs by higher priority jobs until the latest defer time, we reduce the overhead caused by preemptions while still ensuring every job meets its deadline. We present our ongoing work which modifies dual priority of Fixed-Priority scheduling using the worst case response time. Thus we find each task's latest defer time by subtracting its deadline from its worst case response time offline. Then for each job's release time, we adjust the latest defer time online. In this manner, we can then reduce certain amount of preemptions hence reduce the overhead.

1. Introduction

In real-time systems, correct behavior depends not only upon logical correctness, but also upon temporal correctness. In these systems, all jobs have deadlines. In a hard real-time system, each job must be completed no later than its deadline. Even so, there may be times when we might deliberately delay the execution of a job provided it will not cause a deadline to be missed. In this paper, we present an off-line analysis with modified Fixed-Priority scheduling using Dual Priority. This method originally introduced by Davis, et al. [1], calculates a duration during which we can *safely* delay a preemption. We propose an online method of improving their work by adjusting the preemption defer interval as tasks release new jobs.

Preemption occurs when a higher priority task arrives while a lower priority task is executing. In much of the literature, the cost of preemption is assumed to be included within the worst case execution time (WCET) (or sometimes ignored altogether). However, in actual implementations, preemption does incur overhead. These costs occur for two reasons. First, registers of the currently executing (low priority) task must be stored to ensure proper operation upon resumption of execution. Second, the high priority task may cause the contents of the cache to be overwritten, leading to cache misses that would not have occurred without the preemption. Davis, et al., proposed to incorporate deferred preemption into existing scheduling algorithms. The goal is to defer enough that the low priority job finishes executing before the preemption occurs. We propose to extend the duration of the deferred preemption to increase the likelihood that the preemption will no longer be necessary.

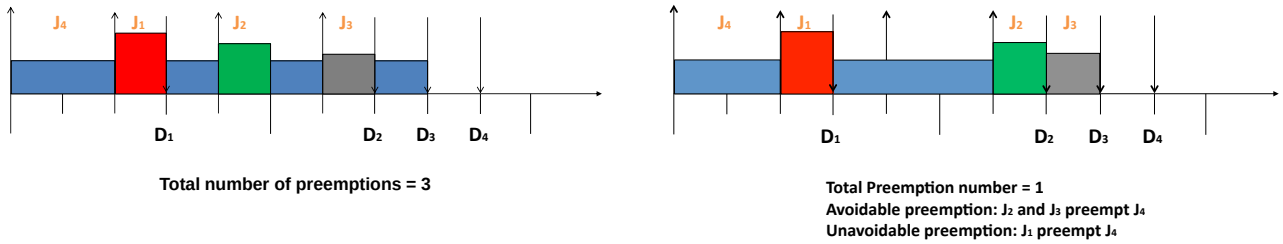
The example below illustrates while some preemptions cannot be avoided without causing deadlines to be missed, others could be avoided.

Example 1. *Consider a system of four jobs J_1, J_2, J_3 and J_4 that arrive at times 2, 4, 6 and 0, respectively, and have deadlines 3, 7, 8 and 9, respectively. J_4 executes for 5 time units and jobs J_1, J_2 and J_3 each execute for 1 time unit. Figure 1 illustrates two schedules for these jobs. Figure 1(a) shows that the preemptive schedule causes 3 preemptions: J_1, J_2 and J_3 each preempt J_4. The first preemption is unavoidable — J_1 must execute as soon as it arrives. However, the preemptions caused by J_2 and J_3 can be avoided without missing any deadlines. Figure. 1(b) illustrates that by deferring those jobs J_2 and J_3, we can reduce the number of preemptions while maintaining feasibility.*

□

In Robert Davis and Andy Wellings [1], they presented the dual priority scheduling algorithm and find the worst case response time by Time Demand Analysis (TDA) [2]. We extend their work to calculate the worst case response time by checking the worst case arrival pattern at each job's release time. Hence we want to make the deferral interval as long as possible but without missing any deadlines.

The remainder of this paper is organized as follows. Section 2 introduces our model and defines important terms to be used in this paper. Section 3 recaps the related work done by Davis et al. Section 4 introduces our extension work for online adjustment. Finally, Section 5 concludes our work and discusses future avenues of research.



(a) Preemptive schedule.

(b) Deferred preemptive schedule.

Figure 1. Two valid schedules.

2. Model and Definitions

We consider a method of reducing preemptions when scheduling periodic [3] or sporadic [4], [5] tasks. Below we introduce the terms and notation we will be using in the following sections.

Periodic and sporadic tasks: The periodic [3] and sporadic [4], [5] task models have proven very useful for the modeling and analysis of real-time computer application systems. In this model, real-time processes recur at regular intervals. Each periodic or sporadic task T_i is characterized by two parameters — a worst case execution requirement (e_i) and a period (p_i). Accordingly, we will model a real-time system $\tau \equiv \{T_1, T_2, \ldots, T_n\}$ as being comprised of a collection of n periodic or sporadic tasks.

In this paper, we will assume that all the system parameters — the number of tasks in the system, the execution requirement and period parameters of each task — are a priori known. Each periodic or sporadic task T_i generates an infinite sequence of jobs $T_{i,0}, T_{i,1}, \ldots, T_{i,k}, \ldots$. We denote the arrival time and deadline of a job $T_{i,k}$ to be $a_{i,k}$ and $d_{i,k}$, respectively. A job does not necessarily start at its arrival time. We let $s_{i,k}$ denote the time when job $T_{i,k}$ first starts to execute. A periodic task $T_i = (p_i, e_i)$ with execution requirement parameter e_i and period parameter p_i generates a job $T_{i,k}$ that arrives at time $a_{i,k} = k \cdot p_i$. Each job $T_{i,k}$ needs to execute for e_i units of time by its deadline of $d_{i,k} = (k+1) \cdot p_i$, for all non-negative integers k. For sporadic tasks, p_i indicates the *minimum* amount of time between two jobs. If a sporadic task generates a job $T_{i,k}$ at time $a_{i,k} = t$, then that job needs to execute for e_i units of time by its deadline of $d_{i,k} = t + p_i$, and T_i's next job $T_{i,k+1}$ can arrive no earlier than $t + p_i$ — i.e, $a_{i,k+1} \geq a_{i,k} + p_i$ for all $k \geq 0$.

For both periodic and sporadic tasks, we denote the worst case response time for each task T_i as r_i.

Latest Defer Time: The latest defer time, Λ_i, for a task τ_i is the latest time τ_i can defer (relative to its arrival time) and ensure it will meet its deadline d_i – i.e., if τ_i defers more than Λ_i, there exists some arrival patterns that will cause τ_i to miss its deadline. We denote each job $T_{i,k}$'s *latest start time* $\lambda_{i,k}$ — thus, $\lambda_{i,k} = a_{i,k} + \Lambda_i$.

We assume a preemptive schedule. Thus, if a job arrives while another job is executing, if the newly arrived job has a higher priority it can interrupt (or preempt) the currently executing job. The costs associated with preemption are generally assumed to be included in the tasks worst case execution times. We let $\boldsymbol{T}_i^{hp}$ denote those tasks whose priority are the same or higher than task T_i, i.e., $\boldsymbol{T}_i^{hp} = \{T_j \mid j \leq i\}$. Similarly, the tasks whose priority are lower than task T_i are denoted $\boldsymbol{T}_i^{lp}$ where $\boldsymbol{T}_i^{lp} = \{T_j \mid j > i\}$.

Deferred preemption intervals: This paper introduces a strategy for reducing the number of preemptions by delaying preemptions when higher priority jobs arrive. We discuss how to incorporate this strategy into Fixed-Priority scheduling algorithm. We define deferral durations as follows.

Let $\tau = T_1, T_2, \ldots, T_n$ be a task set. Assume task T_r is executing and task T_x generates a higher priority job at time t. We define the latest defer time Λ_x as longest deferal preemption interval that lower priority jobs can execute. I.e., the algorithm opts to allow task T_r to continue to executing for an additional Λ_x time units even though T_x has higher priority. When this occurs, we say T_x defers its preemption of T_r, or, alternatively, T_r forces T_x to defer its preemption.

Clearly, we would like to make Λ_x large enough that task T_r is able to finish executing before task T_x has to preempt. Thus, we want the value of Λ_x to be as large as possible but *not* cause any deadlines to be missed.

3. Dual Priority Scheduling

Our goal is to reduce preemptions without sacrificing schedulability. The approach we pursue is to determine the value of Λ_x, a *safe* amount of time to defer preemptions (i.e., a duration that will not cause any deadlines to be missed). We begin by discussing how to find the latest defer time.

We say a job $T_{i,h}$ is in elevated state if $s_{i,h} \geq \lambda_{i,h}$. We first present Fixed-Priority with Dual Priorities introduced by Davis et al. Once $T_{i,k}$ has deferred its preemption by

Λ_i amount of time, it becomes more urgent than it used to be. Hence, we use the *dual priority* scheduling method. When a higher priority task T_x allows a lower priority task to execute for its maximum defer length Λ_x it becomes move urgent that T_x must execute. If this occurs we elevate T_x's priority. In this method, we use an *elevated flag* to indicate if a task has higher priority than its original priority. The tasks in elevated state have higher priority than the tasks in their original *unelevated* state, for instance: T_1 normally has higher priority than T_2. However if T_2 is in elevated state and T_1 is not, then T_2 has higher priority than T_1. If a task has not started to execute before the end of its deferral interval, then its elevated flag is set, giving it higher priority. After the elevated task has completed then the *elevated flag* will set back to false.

In Section 3.5 of [1], Davis et al. have analyzed the latest defer time (which they call the priority promotion time). Let Λ_i be the latest defer time for task τ_i, r_i is the worst case response time and d_i is the deadline of T_i. Then Davis et al. [1] showed $\Lambda_i + r_i = d_i$. Furthermore, there exists a scenario in which τ_i finishes exactly at its deadline when its priority is elevated Λ_i time units after it arrives. The worst case arrival pattern during lower priority task's execution window has also been analyzed by Davis et al. in [1] section 3.2 by using Time Demand Analysis (TDA). With this in mind, let Λ_i be the latest defer time length for τ_i. A critical instant occurs for T_i if during $[a_i, \lambda_{i,k}]$ only tasks in $\boldsymbol{T}_i^{lp}$ or tasks with elevated priority execute and at time $\lambda_{i,k}$ all tasks in $\boldsymbol{T}_i^{hp}$ have their priorities elevated. The following example illustrates the concept of extending deferral time.

Example 2. *Consider a system of four tasks T_1, T_2, T_3 and T_4 shown in Figure 2. The first and the last jobs of T_1 do not execute while T_3 is active. Therefore, we can move T_3s elevation point from 6 to 7, allowing T_4 to complete without being preempted. The worst case response time $r_3 = 14 = \lceil \frac{14}{6} \rceil \cdot 1 + \lceil \frac{14}{10} \rceil \cdot 2 + 7$.*

□

4. Extending the priority elevation point

The TDA equation in [2], $w_i(t) = \sum_{j=1}^{i} c_j \cdot \lceil t/T_j \rceil$ gives the cumulative demands on the processor made by higher priority tasks over [0,t]. If r is the WCRT then $w(r) = r$. Notice that $\lceil \frac{p_i}{p_j} \rceil$ measures the number of times T_i interferes with T_j — i.e., T_i executes while T_j is in the wait queue. By using TDA to determine the worst case response time r_i for each task, we can then find the corresponding Λ_i. TDA certainly shows how many interferences between higher priority tasks and lower priority tasks. However the calculation may be too pessimistic. There exist arrive patterns in which some of the interferences of higher priority tasks do not need to be counted against a task's worst case response time. Specifically, let $[\lambda_{j,k}, d_{j,k}]$ be the interval during which T_j is in elevated state.

We observe that the number of times that T_i interferes with T_j's execution during this interval can be reduced whenever two conditions hold. Let $X_{i,j}$ be the number of times T_i interferes with T_j in the calculation of r_i. If we can be certain that current conditions will result in fewer then $X_{i,j}$ interferences, then we can increase λ_j. If the first and last job of T_i that overlap with $[\lambda_j, d_j]$ execute outside of that interval then we can be *sure* that the response time of $T_{j,k}$ will be no more than $r_j - e_i$. Specifically, let $T_{i,h1}$ and $T_{i,h2}$ be the jobs of T_i that overlap with $\lambda_{j,k}$ and $d_{j.k}$ respectively. If $T_{i,h1}$ executes before $\lambda_{j,k}$ and if $T_{i,h2}$ does not devote its priority before $d_{j,k}$ then $\lambda_{j,k}$ can be increased by e_i. The example below shows the idea by shifting higher priority tasks could reduce the number of interference.

Example 3. *Consider the following example shown in Figure 3, a system of 2 tasks: $T_i = (9, 4)$ and $T_j = (15, 6)$. The number of interferences during [13, 28] as illustrated is only one (because the first and last interferences can be removed), however, by using TDA after taking ceiling the number of interferences are three instead.*

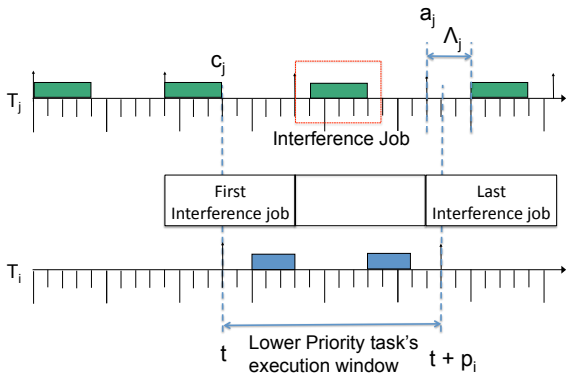
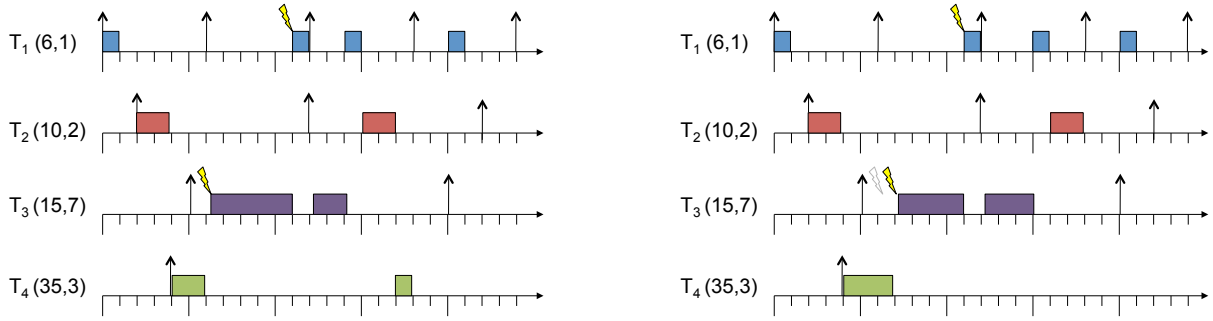


Figure 3. Adjusting response time based on task arrivals

□

Hence we perform an online adjustment in each job's release time for every task. For each higher priority task, if both circumstances above hold we can reduce the number of interference by one. Which allows us to increase the defer time.

To implement this, we add two new arrays-Arr that contain the arrival time of the current job of each task and Comp is a Boolean array which is true for each task whose current job has finished executing. In addition to Λ_i, each task T_i has an associated array $[x_{i,1}, x_{i,2}, ..., x_{i,i-1}]$



(a) Deferral Time without extended

(b) Deferral Time with extended

Figure 2. Extending Deferral Time

containing the number of times the higher priority tasks interfere with T_i when T_i suffers its worst case response time. This is illustrated in Figure 3. and the algorithm is also presented in Algorithm 1. The algorithm is called once at time $\lambda_{i,k}$. Once the defer interval his been extended, Algorithm 1 is not called again.

Algorithm 1 Online Adjustment of $\lambda_{i,k}$

for all higher priority task $T_j, \forall 1 < j \leq i-1$ **do**
  if comp[j] and (Arr[j]+$x_{i.j} \cdot p_j + \Lambda_j \geq d_{i,k}$) **then**
    $\lambda_{i,k} = \lambda_{i,k} + e_j$
  end if
end for

5. Summary and Future Work

In this paper we have presented a method of extending dual priority scheduling with Fixed-Priority scheduling algorithm. When a job's priority is about to be elevated we apply our online adjustment method to increase the defer time. In future we will simulate our online adjustment method to have experimental data for comparing the result with original. In Bril's [6] work, their offline analysis demonstrates that the worst case response time of a higher priority job blocked by lower priority job is based on the amount of time that a task τ_x with a lower priority executes *non-preemptively*. As to our ongoing work is an online extension approach. We would like to extend our work to dynamic-priority scheduling algorithm, such as EDF as well. We also hope to apply this work to systems executing on multiprocessors.

References

[1] R. Davis and A. Wellings, "Dual priority scheduling," in *IEEE Real-Time Systems Symposium*, Dec. 1995, pp. 100–109.

[2] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proceedings of the Real-Time Systems Symposium - 1989*. Santa Monica, California, USA: IEEE Computer Society Press, Dec. 1989, pp. 166–171.

[3] C. L. Liu and 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.

[4] M. Dertouzos and A. K. Mok, "Multiprocessor scheduling in a hard real-time environment," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1497–1506, 1989.

[5] M. Dertouzos, "Control robotics : the procedural control of physical processors," in *Proceedings of the IFIP Congress*, 1974, pp. 807–813.

[6] R. J. Bril and W. F. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited," in *Proceedings of the 19th Euromicro Conference on Real-Time Systems*. IEEE Computer Society Washington, DC, USA, 2007.

Real-time Scheduling of periodic tasks in a monoprocessor system with rechargeable energy storage

Maryline Chetto and Hussein El Ghor
IRCCyN - University of Nantes
1 Rue de la Noë, F-44321 Nantes FRANCE
maryline.chetto@univ-nantes.fr, elghorh@irccyn.ec-nantes.fr

***Abstract*—We are interested in a real-time computing system that is powered through a renewable energy storage device. In this context, two constraints need to be addressed: energy and deadlines. Classical task scheduling, in particular Earliest Deadline First, only accounts for timing parameters of the tasks and consequently is not suitable when considering energy constraints. We show here how to modify Earliest Deadline so as to account for the properties of the energy source, capacity of the energy storage as well as energy consumption of the tasks. We present a scheduling framework called EDeg (Earliest Deadline with energy guarantee) and an exact feasibility test that decides for periodic task sets, whether they can be scheduled without deadline violations. To this end, we introduce the concepts of energy demand and slack energy.**

***Keywords*-scheduling; periodic tasks; earliest deadline; renewable energy;**

I. Introduction

The problem of scheduling tasks on one processor to meet deadlines and energy constraints has been the focus of great interest for about ten years. However, few papers have been devoted to emerging harvesting systems which need to operate perennially thanks to the environmental energy. A key consideration that affects power management in an energy harvesting system is that instead of minimizing the energy consumption and maximizing the lifetime achieved as in classical battery operated devices, the system operates in an energy neutral mode by consuming only as much energy as harvested.

The system we target here consists of a processing unit, an energy harvester such as a solar panel or a furl cell, and a rechargeable energy storage such as a battery or a supercapacitor. We consider a single processor that has to execute a set of independent periodic tasks.

So the problem we have to deal with is: How can we schedule the tasks so as to guarantee their timing constraints perpetually by suitably exploiting both the processor and the available ambient energy.

The goal of this work is first to construct an optimal scheduling algorithm and second, to provide an exact schedulability test. Finally, we show how to dimension the capacity of the storage unit, provided that the worst case recharging rate of the storage unit is enough for ensuring neutral operation.

II. Related works

In [1], the authors are interested in the problem of scheduling periodic tasks in the so-called frame-based systems with a rechargeable battery. In this model, all task periods are identical, all task deadlines are equal to the common period. Consequently, the order of task execution within a frame is not crucial for whether the task set is schedulable or not. Moreover, the power scavenged by the energy source is assumed to be constant and all tasks consume energy at a constant rate. A solution is presented that schedules tasks in such a way that the wasted recharging energy is minimized and the battery level is at all times within two limits, starting with a battery fully charged. The idea behind this algorithm is to insert as little idle time as necessary for recharging the battery and minimizing the length of the schedule. This work is certainly the first one to concentrate on a rechargeable system with hard real-time constraints. However, the solution only deals with frame based systems under the restrictive hypothesis that each task is characterized by an instantaneous consumption power which is constant along time.

More recently, in [7] the so-called LSA scheduling algorithm was proved to be optimal under a more generalized model including hard deadline tasks, periodic or not. LSA is a variation of the famous Earliest Deadline First scheduler: the system starts executing a task only if the task is ready and has the earliest deadline among all ready tasks and the system is able to keep on running at the maximum power until the deadline of the task. In that work, the consumption power of the computing system is characterized by some maximum value which implies that for every task, its total energy consumption is directly connected to its execution time through the constant power of the processing device.

However, in practice, the total energy which can be consumed by a task has no correlation with the worst case execution time [6]. For every task, the worst case instantaneous consumption power depends on the circuitry that is used by the task. It clearly appears as impractical to determine the energy consumption of a task from the worst case consumption power for the computing system. While it is easy to determine the average consumption power of a task

(given by the execution time and the energy consumption), this parameter is of no interest in the so-called hard real-time applications.

Furthermore, some recent studies focused on how to precisely compute the energy that is consumed by a program, independently of the average or the worst case consumption power of the computing system. More particularly, in [6] a bound on worst-case energy consumption of a task is computed through a static analysis by estimating an upper bound on the energy consumption of all individual basic blocks that make up the task.

III. Model and terminology

A. Task Set

We study the case of a Hard Real-Time system which is composed of periodic tasks. The arrival times, energy demands and deadlines of these tasks are known in advance. Such a periodic task set can be denoted as follows: $\tau = \{\tau_i, i = 1, \ldots, n\}$. A four-tuple (C_i, E_i, D_i, T_i) is associated with each τ_i. In this characterization, task τ_i makes its initial request at time 0 and its subsequent requests at times kT_i, $k = 1, 2, ...$ called release times. The least common multiple of $T_1, T_2, \ldots, T_n$ (called the hyperperiod) is denoted by T_{LCM}. Each request of τ_i requires a Worst Case Execution Time (WCET) of C_i time units and has a Worst Case Energy Consumption (WCEC) of E_i. We assume that the WCEC of a task has no relation with its WCET.
A deadline for τ_i occurs D_i units after each request by which task τ_i must have completed its execution. We assume that $0 < C_i \leq D_i \leq T_i$ for each $1 \leq i \leq n$. We define:

- the processor utilization as $U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$.
- and the energy utilization as $U_e = \sum_{i=1}^{n} \frac{E_i}{T_i}$

A job is any request that a task makes. A four-tuple (r_j, C_j, E_j, d_j) is associated with a job J_j and gives its release time, worst case execution time, worst case energy consumption and (absolute) deadline respectively. A job can be preempted and later resumed at any time and there is no time or energy loss associated with such preemption.

B. Energy

Our system uses an energy storage unit that has a nominal capacity, namely E, corresponding to a maximum energy (expressed in Joules or Watts-hour). The energy level has to remain between two boundaries E_{min} and E_{max} with $E = E_{max} - E_{min}$. If the storage is fully charged, and we continue to charge it, energy is wasted. In contrast, if the storage is fully discharged, no task can be executed.

In order to characterize the energy source, we define the WCCR (Worst Case Charging Rate), namely P_r, which is a lower bound on the harvested source power output. P_r is then the instantaneous charging rate that incorporates all losses caused by power conversion and charging process. We assume that energy production times can overlap with the consumption times.

C. Definitions

- A schedule Γ for τ is said to be *valid* if the deadlines of all tasks of τ are met in Γ, starting with a storage fully charged.
- A task set τ is said to be *temporally-feasible* if there exists a valid schedule for τ without considering its energy constraints.
- A task set τ is said to be *feasible* if there exists a valid schedule for τ with considering its energy constraints.
- A scheduling algorithm will be called *optimal* if it finds a valid schedule whenever one exists.

IV. Background materials

The problem of scheduling periodic tasks on one processor with no energy constraint has been an active area of research for more than thirty years (see, e.g., [3]). In [5], Dertouzos showed that Earliest Deadline First (EDF) is optimal among all preemptive scheduling algorithms. EDF schedules at each instant of time t, the ready task (i.e the task that may be processed and is not yet completed) whose deadline is closest to t. The EDF algorithm is typically preemptive, in the sense that, a newly arrived task can preempt the running task if its absolute deadline is shorter. This dynamic priority assignment allows EDF to exploit the full processor, reaching up to 100% of the available processing time.

In general, the implementation of EDF consists in executing tasks according to their urgency, as soon as possible with no inserted idle time. Such implementation is known as EDS (Earliest Deadline as Soon as possible). Nevertheless, in some applications, it can be preferable to postpone execution of periodic tasks, executing them by the so called EDL (Earliest Deadline as Late as possible) strategy, for example when some additional aperiodic tasks with unexpected arrival times require to be run as soon as possible [4].

V. Feasibility analysis under energy constraints

In order to develop a procedure for the feasibility assessment of a periodic task set with energy constraints, we give some definitions. Let us consider the periodic task set τ and the interval $[t_1, t_2)$.

- The processor demand of τ in $[t_1, t_2)$, is $h(t_1, t_2) = \sum_{D_i \leq t_2 - t_1} \left(1 + \left\lfloor \frac{t_2 - t_1 - D_i}{T_i} \right\rfloor\right) C_i$
- The energy demand of τ in $[t_1, t_2)$, is $g(t_1, t_2) = \sum_{D_i \leq t_2 - t_1} \left(1 + \left\lfloor \frac{t_2 - t_1 - D_i}{T_i} \right\rfloor\right) E_i$

The processor demand (resp. the energy demand) is a measure of how much computation (resp. energy) is requested by all the jobs which have both their release times and their deadlines, in a given interval of time. It is clear

that for a given time length and among all intervals, the initial one has the maximum fraction of processor and energy demanded by the jobs i.e. $h(t_1, t_2) \leq h(0, t_2 - t_1)$ and $g(t_1, t_2) \leq g(0, t_2 - t_1)$.

For simplicity, we respectively denote as $h(t)$ and $g(t)$ the processor demand and the energy demand of the task set τ in $[0, t)$. So, $h(t) = \sum_{D_i \leq t} \left(1 + \left\lfloor \frac{t-D_i}{T_i} \right\rfloor\right) C_i$ and $g(t) = \sum_{D_i \leq t} \left(1 + \left\lfloor \frac{t-D_i}{T_i} \right\rfloor\right) E_i$.

Without energy constraint, the exact schedulability analysis is based on the processor demand criterion and is stated as follows [2]:

Theorem 1. *A task set τ is temporally-feasible if and only if $U_p \leq 1$ and $\forall t > 0, h(t) \leq t$.*

This exact feasibility test is of pseudo-polynomial complexity since the points in which the test has to be performed correspond to deadlines within the hyperperiod T_{LCM}. Suggestions for practical improvements in testing Theorem 1 have been given in [8] and [9].

Here, we extend Theorem 1 to account for energy consumption:

Theorem 2. *A task set τ is feasible if and only if*

- *τ is temporally-feasible,*
- *$U_e \leq P_r$ and $\forall t > 0, g(t) \leq E + P_r t$*

VI. The optimal scheduling algorithm

The intuition behind the scheduling algorithm is to run tasks according to the earliest deadline first rule. However, before authorizing a task to execute, the storage level must be sufficient to provide energy for all future occurring tasks, considering their timing and energy requirements and the replenishment rate of the storage unit. And if this condition is not verified, the processor has to be idle so that the storage unit recharges as much as possible and as long as the system will be able to meet all the deadlines i.e. the system will have available time to remain idle. Following the idea described above, we propose the $EDeg$ (Earliest Deadline with energy guarantee) algorithm. To formally present the algorithm, we need to introduce two concepts:

- The *slack time of the system* at current time t, is the length of the longest interval starting at t during which the processor may be idle continuously while still satisfying all the timing constraints.
- The *slack energy of the system* at current time t, is the maximum amount of energy that can be consumed from t continuously while still satisfying all the timing constraints.

In the following description, t is the current time, $E(t)$ is the *residual capacity* of the storage unit at time t i.e. the energy that is currently stored. $Slack.energy(t)$ and $Slack.time(t)$ are respectively the slack energy and the slack time at time t. PENDING is a boolean which equals true whenever there is at least one job in the ready list queue. We use the function wait() to put the processor to sleep and function execute() to put the processor to run the ready job with the earliest deadline.

The framework of the optimal scheduling algorithm is as follows:

Algorithm 1 Earliest Deadline with energy guarantee algorithm (EDeg)

```
while (1) do
  while PENDING=true do
    while (E(t) > E_min and Slack.energy(t) > 0) do
      execute()
    end while
    while (E(t) < E_max and Slack.time(t) > 0) do
      wait()
    end while
  end while
  while PENDING=false do
    wait()
  end while
end while
```

We notice that:

- $EDeg$ degenerates to an EDS policy if $E_{max} = 0$ and an EDL (Earliest Deadline as Late as possible) policy if $E_{max} = \infty$.
- We never run out of storage (that is, we never dispatch tasks when there is no energy); this is obvious from the algorithm that does not allow tasks to run after E_{min}.
- We start charging the storage unit when, either it is empty or there is no more sufficient energy to guarantee the feasible execution of all future occurring tasks i.e. the system has no more slack energy.
- the charging process aims to charge at the maximum level provided there is sufficient slack time.
- We only waste recharging power when there are no pending tasks and the storage unit is full.

As a consequence, we can prove the following theorem.

Theorem 3. *Algorithm EDeg is optimal.*

VII. Practical considerations

The computations of Slack.energy(t) and Slack.time(t) are thus the keys to the operation of the EDeg algorithm. As shown in [10], the slack time of a periodic task set at a given time instant can be obtained on-line by computing the dynamic EDL schedule, with complexity O(K.n). n is the number of periodic tasks, and K is equal to $\lfloor R/p \rfloor$, where R and p are respectively the longest deadline and the shortest period of current ready tasks.

The slack energy at time t is computed only when there is at least one job, say J_j which will be released after t and has a deadline d_j that is less than or equal to that of the

highest priority job, ready at t. For such job, we compute Slack.energy(J_j,t), given by $E(t)+P_r(d_j-t)-A_j$ where A_j is the processor demand within $[t, d_j)$. Slack.energy(J_j,t) clearly represents the amount of energy surplus in the storage that can be used from t until the start time of J_j while still guaranteeing its timing and energy requirements. The slack energy of the system is determined by the minimum slack energy of all the jobs. The complexity for computing the slack energy is O(K.n) too.

A suggestion to improve the efficiency of the scheduler in terms of overhead is to compute statically a lower bound on the slack time and a lower bound on the slack energy and use them instead of exact values which are computed on-line. The effect will be only, first to stop charging earlier and second to stop executing tasks earlier. As a consequence, decreasing the processor overheads due to computations will cause increasing the number of tasks preemptions.

VIII. Illustrative Example

Consider a task set τ that is composed of the tree following tasks: $\tau_1(2, 16, 7, 20)$, $\tau_2(2, 10, 4, 5)$ and $\tau_3(1, 6, 9, 10)$. The storage capacity is $E = 10$ and we assume that $E_{min} = 0$ and $E_{max} = E$. The recharging power P_r is constant and equal to 4. To simplify the illustration, we assume that tasks consume energy at constant rate. We note that the processor utilization and the energy utilization are respectively 0.6 and 3.4, consequently no more than 1 and 4. By scheduling the task set τ according to EDF on the first hyper-period i.e. from 0 to 20, we can verify that τ is temporally-feasible. The schedule which is produced by the EDeg scheduler for τ in the first hyper-period is described on *(Figure 1)*. Let us explain how this schedule is constructed in the first steps.

At time 0, the storage is full. τ_2 is the highest priority task, executes until time 2 and consumes 10. At time 2, $E_2 = 8$. τ_1 is the highest priority task and the slack energy is undefined (no job released after 2 with deadline less than 4). τ_1 executes completely until time 4 and consumes 16. 8 units of energy are produced. At time 4, $E_4 = 0$. The processor has to remain idle as long as the storage has not refilled it (predicted at time 6.5) and the latest start time has not been attained (at time 6 which is computed using EDL). At time 6, $E_6 = 8$. τ_2 is the highest priority task, executes until time 8 and consumes 10. At time 8, $E_8 = 6$. τ_3 is the highest priority task, executes and completes exactly at time 9 that coincides with its deadline.

IX. Conclusion

We have presented the framework of a monoprocessor preemptive scheduling algorithm, namely EDeg, that is a variation of EDF able to cope with energy constraints. EDeg has been designed to schedule any set of time critical tasks, periodic or not, given any energy source profile with constant power production or not and given an energy storage unit with limited capacity. Proof of optimality and validation of the exact feasibility condition attached to EDeg are the object of a Work in Progress.
This paper specifically focussed on a system that receives energy at constant rate and has to run periodic tasks with non related computation and energy requirements.

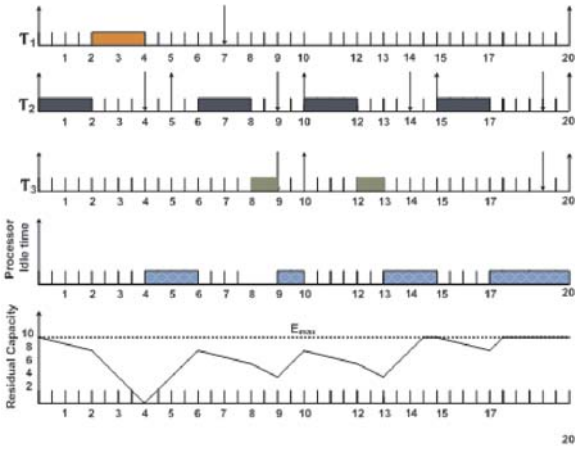


Figure 1. The EDeg schedule

References

[1] A. Allavena and D. Moss, "Scheduling of Frame-based Embedded Systems with Rechargeable batteries", *Workshop on Power Management for Real-Time and Embedded Systems*, 2001.

[2] S.K. Baruah, A.K. Mok and L.E. Rosier, "Preemptively Scheduling Hard Real-Time Sporadic Tasks on One Processor", *Proc. 11th IEEE Real-Time System Symp.*, pp. 182-190, 1990.

[3] G.C. Buttazzo, Hard Real-Time Computing Systems, Springer, 2005

[4] H. Chetto, M. Chetto, "Some Results of the Earliest Deadline Scheduling Algorithm". *In Proceedings of the IEEE Transactions on Software Engineering*, Vol. 15, No. 10, pp 1261-1269, 1989.

[5] M.L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes", *Proc. Int'l Federation for Information Processing Congress*, pp. 807-813, 1974.

[6] R. Jayaseelan, T. Mitra, X. Li, "Estimating the Worst-Case Energy Consumption of Embedded Software," *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pp.81-90, 2006

[7] C. Moser, D. Brunelli, L. Thiele, L. Benini, "Real-time scheduling for energy harvesting sensor nodes", *Real-Time Systems*, Volume 37 , Issue 3, Pages: 233 - 260,December 2007

[8] I. Ripoll, A. Crespo and A.K. Mok, "Improvement in Feasibility Testing for Real-Time Tasks", *Real-Time Systems* 11, 1996.

[9] F. Zhang and A. Burns, "Schedulability Analysis for real-time systems with EDF scheduling", *IEEE Transactions on Computers*, Vol. 58, N 9, September 2009.

[10] M. Silly, "The EDL Server for Scheduling Periodic and Soft AperiodicTasks with Resource Constraints", *Real-Time Systems*, Volume 17 , Issue 1, July 1999.

Network-Aware, Energy-Conscious, Fair Service for Real-Time Applications on Multiprocessor SoC*

Thidapat Chantem†, X. Sharon Hu†, Christian Poellabauer†, Jun Yi† and Liqiang Zhang‡
†Department of CSE
University of Notre Dame
Notre Dame, IN 46556
{tchantem, shu, cpoellab, jyi}@cse.nd.edu

‡Department of CIS
Indiana University South Bend
South Bend, IN 46634
liqzhang@iusb.edu

Abstract

We consider systems consisting of wireless nodes that execute CPU intensive applications on multiprocessor system-on-a-chip (MPSoC) and must transmit packets over the network in a timely manner. Existing methods do not consider packet deadlines in conjunction with energy and real-time task performance, making it hard to predict system behavior. We present an energy-aware adaptive CPU scheduling algorithm to maximize the number of packet deadlines met in a fair manner and discuss future work.

1 Introduction and Related Work

Wireless networks are now common in a variety of applications, e.g., [3, 5]. While many wireless sensor nodes typically require minimum hardware to perform lightweight tasks (e.g., periodically waking up to sense and transmit data), powerful processing nodes can also be found in certain applications for executing computationally intensive tasks and transmitting packets over the network. Example applications are surveillance and mobile gaming systems. In a surveillance system, a wireless node periodically captures a video, processes the frames and transmits them to clients. As for the gaming system, the processor is kept busy with a large number of tasks (e.g., rendering graphics) while a large amount of data is sent to the user's opponents.

To cater to the high computing demand imposed by the applications mentioned above, one alternative is to use high-end, power-hungry, processors. Since wireless nodes are generally battery powered, to save energy and prolong the lifetime of these nodes, multiprocessor system-on-chips (MPSoCs) present a better alternative for wireless nodes requiring higher computational power. More work can be completed by running processor cores in parallel at lower voltage and frequency, thus saving energy. The parallel execution capabilities of such MPSoC-based wireless nodes, however, introduces new challenges in terms of reducing energy consumption while satisfying real-time constraints.

There is a large research body on energy minimization in multiprocessors running real-time applications, e.g., [1, 2], though the majority of the work solely focuses on optimizing real-time task performance without any consideration for packet deadlines. Meeting packet deadlines is important in ensuring performance requirements such as data freshness. At the same time, network-aware work usually focuses on trading network energy with packet latency using packet scheduling and ignores task deadlines [7, 8]. To the best of our knowledge, the work by Yi et al. is the only energy-aware solution that explicitly considers packet deadlines on uniprocessor architectures executing real-time tasks [9]. However, it is unclear how the proposed solution may extend to multiprocessor architectures. Additionally, in the approach of [9], it is difficult, if not impossible, to predict which packets will miss their deadlines. This could lead to unfairness in packet transmission. That is, some tasks may consistently be able to successfully transmit their packets while the packets of other tasks starve.

In this work, we design an adaptive CPU scheduling algorithm that maximizes the number of packets that meet their deadlines in a fair manner. Fairness is used to ensure that each task has an adequate number of successfully transmitted packets relative to their importance. The main idea of our work is to prevent executions of jobs whose packets will be dropped to save CPU energy while allowing specific packets to be sent. Using a network reservation-based approach in [9], network energy is also managed.

2 Preliminaries

We consider a set of n independent periodic real-time tasks. Each task τ_i is described by its worst-case execution

*This work is supported in part by NSF under grant numbers CNS-0834180 and CNS-0834230.

time C_i and period T_i. All tasks are synchronous. The j-th instance (job) of task τ_i is denoted by $\tau_{i,j}$. We assume a partitioned scheduling approach in which tasks are assigned to their respective cores using the algorithm in [2] and that the tasks on each core are schedulable using the algorithm in [6]. No job or task migration is allowed.

Two types of tasks are considered: packet-generating and non packet-generating. Non packet-generating tasks are hard real-time tasks. Without loss of generality, we assume that every job of packet-generating tasks generates a packet at the end of its execution. Packets have firm real-time deadlines, i.e., they must be transmitted by their deadline or they will be dropped. Packets from different instances of the same task are equal in size while packets from different tasks may vary in size. A packet P_j is described by its deadline X_j and worst-case transmission time Z_j.

The MPSoC consists of m homogeneous cores, each of which can run at k discrete speed levels. Cores can independently change speed. We assume that transition overheads associated with switching from one speed to another have been included in the task worst-case execution times.

The processor cores share a network card. Packets are transmitted in an earliest-deadline first (EDF) manner. Transmissions are preemptable at some minimum unit. Since packets that cannot meet their deadlines are dropped, the number of packets transmitted is used interchangeably with the number of packets that meet their deadlines.

Each node uses TDMA-like periodic time slots to send and receive packets. No network communication takes place outside of these time slots. The time slots, described by a period T_{ts} and length C_{ts}, may change over time to reflect different network usage levels. We assume that incoming packets are buffered at the sender and arrive at the beginning of each time slots during which the wireless node avoids transmitting any packets.

We are interested in solving the following problem: Given the real-time task set, MPSoC, network card and transmission models described above, determine an execution pattern for packet-generating jobs such that all non packet-generating jobs meet their deadlines, the number of packets transmitted over the network is maximized in a fair manner, and the energy consumption is minimized.

3 Motivations

We use a simple example to motivate our problem. Assume that we have a set of four tasks as shown in Table 1. In addition, our MPSoC consists of four identical cores, m_1, m_2, m_3, and m_4. Using the approximation algorithm in [2], each core is assigned a task to execute.

For this example, we assume that jobs always require their worst-case execution times, that cores can adjust their speeds in a continuous manner, and that the network card is allowed to transmit packets whenever it wishes. It must be noted, however, that these assumptions are made for ease of explanation and that we do not rely on any such assumption to solve our general problem. Using LaEDF [6], each job finishes its execution right before its deadline. The corresponding processor and network card states are shown in Figures 1(a) and 1(b), respectively.

Table 1. Example Task Set

Task	C	T	Packet?	X	Z
τ_1	1	2	Yes	$T+1$	0.5
τ_2	1	3	Yes	$T+2$	1
τ_3	2	6	Yes	$T+2.3$	1
τ_4	3	12	Yes	$T+2.3$	1

As shown in Figure 1(b), no packets generated by tasks τ_3 and τ_4 were transmitted. This is unfair because all packets generated by tasks τ_1 and τ_2 were sent. What is worse, the same pattern will persist to the future, which means that packets generated by τ_3 and τ_4 will never be transmitted and the energy used to execute instances of τ_3 and τ_4 is wasted.

Now, suppose that we had a mechanism to select jobs to execute in a fair manner. The resultant MPSoC and network card state might be as shown in Figures 2(a) and 2(b), respectively. The total number of packets transmitted does not change but some packets generated by τ_3 and τ_4 can now be transmitted. In addition, the CPU does not waste energy executing jobs whose packets are not transmitted.

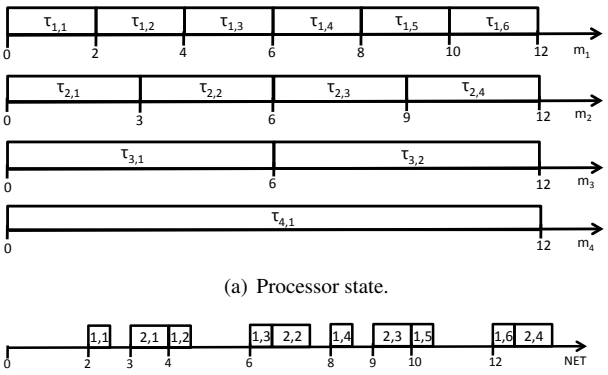
4 Network-Aware Adaptive CPU Scheduling

We provide our general approach, discuss the fairness metric, and give a detailed explanation of our algorithm.

4.1 Overview

The main component of our algorithm is the manager task (MT), which runs on one of the cores, is a periodic task, and competes for resource. Let us define an observation window (ObsWin) to be the least common multiple of the task hyperperiod and T_{ts}. In an ObsWin, the network card monitors the packet transmission pattern and sends this information to the MT, which will then use it to compute the current system fairness level. The MT also compiles a list of tasks whose packets are being transmitted more than it should and send it to the cores. In the next ObsWin, the scheduler on each core avoids executing jobs of these tasks during high interference time intervals (defined below), thus allowing packets from other tasks to be transmitted.

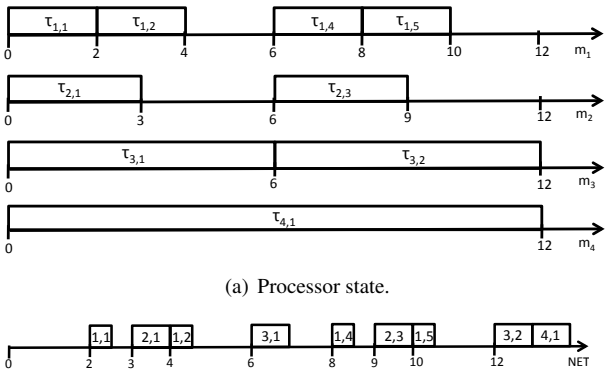
Since cores execute jobs using dynamic voltage scaling (DVS) and some jobs are dropped, processor energy is saved. At the same time, since these jobs are dropped in a controlled manner, packets will be transmitted fairly. Note



(a) Processor state.

(b) Network state, each box indicates packet by $\tau_{i,j}$ is transmitted.

Figure 1. System state without network-aware scheduling.



(a) Processor state.

(b) Network state.

Figure 2. System state with network-aware scheduling.

that we focus on CPU energy and not network energy, as the network energy has already been managed during the negotiation of the periodic time slots [9].

Our algorithm is adaptive in the sense that it may take a number of ObsWins for the packet transmission pattern to stabilize. However, the system can perform exactly as it has in the previous ObsWin once this is the case until, say, a new task joins the system or the period or length of the time slot changes. In such cases, the MT is reactivated until the system stabilizes yet again.

4.2 Fairness Metric

Although our algorithm is independent of the specific fairness metric used, we use Jain's fairness index [4] to measure system fairness level in this work. Jain's fairness index F can be defined as follows.

$$F = \frac{\left(\sum_{\tau_i \in G}(w_i \lambda_i)\right)^2}{|G| \cdot \sum_{\tau_i \in G}(w_i \lambda_i)^2} \leq 1 \qquad (1)$$

where w_i is the weight of task τ_i, G is the set of packet-generating tasks, and λ_i denotes the packet deadline meet ratio of packet-generating task τ_i.

4.3 Algorithm Details

The relevant parts of our proposed algorithm are shown in Algorithm 1, with checkpoints omitted for brevity.

To observe the original system behavior during the first ObsWin (e.g., where packet deadline misses occur, if any), all jobs are executed using LaEDF and the network card transmits as many packets as possible while keeping track of the deadline miss ratios $\boldsymbol{\lambda}$, as well as $\mathbf{HT}$, which is a list of high interference time intervals $[t_s, t_e]$ where t_s is the release time of a packet whose deadline is missed and t_e is its deadline. If there is no packet deadline miss, our algorithm is not activated and the system continues as before.

Using $\boldsymbol{\lambda}$, the MT computes F as in (1). The objective of the MT is to improve the system fairness level by identifying tasks which have had an unfair access to the network. That is, we wish to reduce the number of packets transmitted by such tasks so that the number of packets transmitted by other tasks (which have been disadvantaged) can be increased. Specifically, we use two embedded loops that iterate through packet-generating tasks, one following the non-increasing order of meet ratios and the other in reverse. The goal is to find pairs of tasks, one with a higher meet ratio and another with a lower meet ratio, whose packets will likely interfere with one another. The MT then determines how the system fairness level will change should the number of packets of the task with a higher (lower) meet ratio is decreased (increased) by one. If the system fairness level will increase, the task with the higher meet ratio is added to the set $\mathbf{TL}$, which will be used by the scheduler on each core to determine which jobs to drop.

As for the scheduler on each core, its only duty is to determine whether to execute the next ready job. This is accomplished by examining whether the associated task of the current job appears in $\mathbf{TL}$ and determining whether the job deadline falls within one of the high interference time intervals. Here, the idea is to balance energy consumption with useful work completed by the processor cores.

5 Summary and Future Work

We proposed an energy-aware adaptive CPU scheduling algorithm to maximize the number of packets transmitted

Algorithm 1 Energy-Aware Adaptive CPU Scheduling

Upon end of each ObsWin
  if at least one packet missed its deadline **then**
    execute MT with $\boldsymbol{\lambda}$ and $\mathbf{HT}$ from network card
Upon each execution of MT
  compute F // current fairness level of system
  $R \leftarrow$ packet-generating tasks sorted in a non-increasing order of $\mathbf{w}\boldsymbol{\lambda}$
  $R' \leftarrow inverse(R)$
  for each task $\tau_i \in R$ **do**
    $\lambda_i \leftarrow \frac{\text{num\_packets\_transmitted}_i - 1}{\text{num\_packets\_generated}_i}$
    for each task τ_j in R' **do**
      if τ_i.deadline **and** τ_j.deadline $\in \mathbf{HT}$ **then**
        $\lambda_j \leftarrow \frac{\text{num\_packets\_transmitted}_j + 1}{\text{num\_packets\_generated}_j}$
        compute F' // using new values for λ_i and λ_j
        if $F' > F$ **then**
          $\mathbf{TL} \leftarrow \mathbf{TL} \cup \tau_i$
          $F \leftarrow F'$
  send $TL(m)$ and $HT(m)$ to m, for all cores m
Upon each scheduling point on core m
  $j \leftarrow$ next ready job // using LaEDF
  if j.deadline $\in HT$ **and** j.getTask $\in TL$ **then**
    drop j

over the network in a fair manner for real-time applications running on MPSoCs. Our algorithm is independent of the fairness metric used and requires minimum interactions between the network card and the processor cores.

As this work is ongoing, there are still many improvements to be made and several challenges to be solved. For instance, the observation window is currently defined to be the least common multiple of the task hyperperiod and T_{ts}. While this definition of ObsWin allows the MT to easily determine which jobs should not execute and when, the actual length of ObsWin in practice may be too large since it is a function of the task hyperperiod. Using a large ObsWin has several drawbacks. For instance, changes may take place very slowly, prohibiting the system from responding to dynamic perturbations in a timely manner. Also, a long ObsWin entails that the size of $\mathbf{HT}$ will be large, requiring a large amount of memory space and causing long latency when executing our algorithm.

In the current version of the algorithm, some jobs are dropped even if their packets may in reality not interfere with the other packets because of the high variation in job execution times. In other words, the current algorithm may too aggressively drop jobs. A heuristic is needed to determine whether such jobs should be executed based on their expected execution times and network state. Also, job migration may help to further improve the system fairness level, as well as energy saving, and is worth exploring.

An implicit philosophy behind the proposed algorithm is the minimization of the interactions between the network card and processor cores, as a constant update from the network card regarding its status can incur unacceptable overheads. However, due to the lack of constant communications, the cores do not have full knowledge of the network state, which could greatly help in improving performance. At the same time, the use of the MT requires that both $\mathbf{TL}$ and $\mathbf{HT}$ be shared, possibly via cache. This means that our algorithm may require that the memory management unit arbitrates the access of these shared information (and possibly causing delays in some job executions when the MT is executing). Determining the "right" amount of status update from the network card and the best way to share information among cores are subject to ongoing investigation.

Finally, once all the above challenges have been addressed, we plan on evaluating our work against the work in [9] for uniprocessor architectures and against the state-of-the-art energy-aware algorithm that does not consider packet deadlines for MPSoC cases.

References

[1] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proc. of the Int. Symp. on Parallel and Distributed Processing*, page 113.2, Apr. 2003.

[2] J.-J. Chen, C.-Y. Yang, H.-I. Lu, and T.-W. Kuo. Approximation algorithms for multiprocessor energy-efficient scheduling of periodic real-time tasks with uncertain task execution time. In *Proc. of the Real-Time and Embedded Technology and Applications Symp.*, pages 13–23, Apr. 2008.

[3] T. H. et al. VigilNet: An integrated sensor network system for energy-efficient surveillance. *Trans. on Sensor Networks*, 2(1):1–38, Feb. 2006.

[4] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical report, DEC Research Report, 1984.

[5] A. Mainwaring, J. Polastre, R. S. D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proc. of the Int. Workshop on Wireless Sensor Networks and Applications*, pages 88–97, Sept. 2002.

[6] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. *Operating Systems Review*, 35(5):89–102, Oct. 2001.

[7] V. Raghunathan, S. Ganeriwal, C. Schurgers, and M. Srivastava. E2WFQ: An Energy Efficient Fair Scheduling Policy for Wireless Systems. In *Proc. of the Int. Symp. on Low Power Electronics & Design*, page 30.

[8] E. Uysal-Biyikoglu, B. Prabhakar, and A. E. Gamal. Energy-efficient packet transmission over a wireless link. *Trans. on Networking*, 10(4):487–499, Aug. 2002.

[9] J. Yi, C. Poellabauer, X. Hu, J. Simmer, and L. Zhang. Energy-conscious co-scheduling of tasks and packets in wireless real-time environments. In *Proc. of the Real-Time and Embedded Technology and Applications Symp.*, pages 265–274, Apr. 2009.

Leakage-Aware Real-Time Scheduling For Maximal Temperature Minimization

Gang Quan
Electrical and Computer Engineering Department
Florida International University
gang.quan@fiu.edu

Shangping Ren
Department of Computer Science
Illinois Institute of Technology
ren@iit.edu

***Abstract*—Thermal management problem has become a prominent issue as power consumption continues to grow exponentially. The leakage/temperature dependency becomes critical in power and thermal aware design as the processor continues to evolve into the the deep sub-micron domain. This paper seeks to explore fundamental principles in thermal aware design when taking the leakage/temperature dependency into considerations. We show and formally prove that, under certain realistic conditions, using the lowest constant processor speed that can guarantee deadlines of all real-time tasks is an optimal method to minimize the maximal temperature for a real-time system. We also use empirical results to justify the validation of this conclusion. We then discuss the possible future extension of this work.**

I. Introduction

The power consumption of the processors has been growing exponentially with each technology generation, and is expected to continuously grow rapidly in the future [1]. The soaring power consumption of processors has posed challenges not only on how to provide enough power source for a system, and but also how to manage the heat generated by the system. The escalating heat has directly led to high packaging and cooling costs, and threaten to significantly degrade the performance, life span, and reliability of computing systems, or even cause the system to fail [2], [3]. Therefore, as processors power consumption continues to rise, the thermal management problem has become an ever increasingly critical issue in the design of computing systems.

As semiconductor technology continues to scale down, the leakage plays a more and more important role [4], [5]. This is particularly true since the leakage power consumption is comparable or even dominates the dynamic power consumption in the deep sub-micron IC circuits. High power consumption causes high temperature, and high temperature increases leakage power and thus the overall power consumption. A thermal-conscious or power-conscious technique becomes ineffective if this temperature/leakage relation is not properly addressed in the deep sub-micron domain.

While reducing power consumption in general helps to lower the temperature, the temperature-constrained scheduling problem is drastically different from the energy-aware scheduling problem, as evidenced in recent studies [6], [7], [8], [9]. Therefore, new guidelines and principles on thermal aware computing need to be developed. Taking the leakage/temperature dependency into considerations makes the thermal aware design problem even more complex.

Consider a real-time job with deadline of $t = D$ and execution time of E. A well-known principle to reduce the energy, as shown by schedule S_1 in Figure 1, is to apply the lowest constant speed (i.e. v_0) within the entire interval so that the task just meets its deadline. Note that, when the leakage is taken into consideration [10], [11], S_1 is not necessarily optimal in terms of the overall energy reduction. In addition, previous researches [7], [8], [12] have shown that an optimal solution for energy minimization is not necessarily the optimal solution for peak temperature minimization. It is very suspicious that such a schedule has the lowest peak temperature. Alternative schedules include the one (i.e S_2) that first runs with a lower speed (i.e $v_1 < v_0$) and then a lower speed (i.e $v_2 > v_0$), or vice versa (S_3). When considering the leakage/temperature dependency, each schedule seems to have its own reasons to decrease or increase the maximal temperature. Then, the questions are: How should we execute the task judiciously such that the maximal temperature within the interval can be minimized? Are there any general guidelines that we can follow or we will have to deal with different scenarios case by case?

In this paper, we show that, under some realistic conditions, using the constant speed is the best way to minimize the peak temperature within an interval. We formulate this conclusion as a theorem and formally prove it. We also use empirical results to justify the conditions in the theorem. In the rest of the paper, Section II introduces system models and motivates our research. Section III presents our theorem and proof, as well as the empirical results to justify our theorem. We draw conclusions and point out our future work in Section IV.

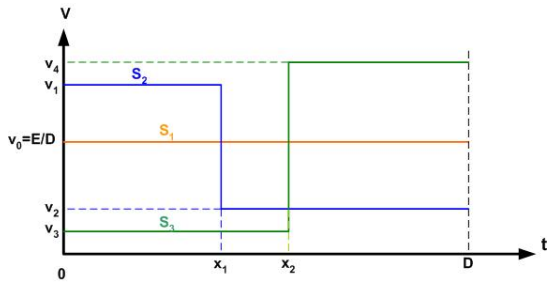


Fig. 1. Three schedules for a job set with deadline D and total execution time E.

II. System Model

We consider a real-time application consisting of n jobs, i.e. $\mathcal{J} = \{J_0, J_1, \cdots, J_{n-1}\}$, and all jobs have a common deadline D. Each job J_i has a worst case execution cycle of e_i, and the total workload of the job set is denoted as E. Since all jobs have the same deadline, we can equivalently treat the model as a single job with deadline D and work load E, and $E = \sum_{i=0}^{n-1} e_i$.

A. Thermal Model

The thermal model used in our paper is similar to that in Shadorn et al. [13]. Specifically, assuming a fixed ambient temperature (T_{amb}), let $T(t)$ be the temperature at time t, and we have

$$R_{th}C_{th}\frac{dT(t)}{dt} + T(t) - R_{th}P(t) = T_{amb}, \tag{1}$$

where $P(t)$ denotes the power consumption (in $Watt$) at time t, and R_{th}, C_{th} denote the thermal resistance (in $J/^oC$) and thermal capacitance (in $Watt/^oC$), respectively. We can then scale T such that T_{amb} is zero and get

$$\frac{dT(t)}{dt} = aP(t) - bT(t), \tag{2}$$

where $a = 1/C_{th}$ and $b = 1/R_{th}C_{th}$. For the rest of the paper, we assume that the initial temperature for the processor equals to its ambient temperature.

B. Power Model

According to Liao et al. [4], the leakage power can be estimated by

$$P_{leak} = N_{gate} \cdot I_{leak} \cdot v \tag{3}$$

where N_{gate} is the total number of gates, I_{leak} is the leakage current, v is the supply voltage, and

$$I_{leak} = I_s \cdot (A \cdot T^2 \cdot e^{((\alpha \cdot v + \beta)/T)} + B \cdot e^{(\gamma \cdot v + \delta)}) \tag{4}$$

where I_s is the leakage current based on a pre-determined reference temperature and supply voltage, T is the system's operating temperature, and A, B, α, β, γ, and δ are technology dependent constants. Some researches, such as that by Bao et al. [14], employ equation (4) directly to capture the leakage/temperature dependency in scheduling analysis. However, due to the non-linear and high-order magnitude terms in equation (4), such a model or tool can be too complex and cumbersome to be used for more rigorous real-time analysis and scheduling technique development.

Liu et al. [12] showed that, for a given supply voltage, the leakage changes with temperature super linearly. Based on this observation, a number of researches (such as [15], [16]) adopt a simple temperature/leakage linear model that assumes the leakage current changes linearly *only* with temperature. However, as can be seen from equation (4), leakage varies not only with temperature but also supply voltage as well. We thus approximate the leakage power for a processor with the following linear function

$$P_{leak}(t) = c_0 v(t) + c_1 T(t) \cdot \tag{5}$$

where c_0 and c_1 are constants, and $v(t)$ is the supply voltage at time t. Constants c_0 and c_1 can be determined by curve fitting based on equation (4). As can be seen from equation (5), we model the leakage such that it changes with both the temperature and supply voltages.

For dynamic power, we assume [17]

$$P_{dyn}(k) = c_2 \cdot v^3(t) \tag{6}$$

c_2 is also a constant and can be determined through profiling. Based on equation (5), (6), and (2), we have

$$\frac{dT(t)}{dt} = A(v(t)) - BT(t) \tag{7}$$

where

$$A(v(t)) = a(c_0 v(t) + c_2 v^3(t)) \tag{8}$$

and

$$B = (b - ac_1). \tag{9}$$

Furthermore, if the processor runs at a constant speed $v(t) = v$ during the interval $[t_0, t_e]$, let the starting temperature be T_0, by solving equation (2), the ending temperature can be formulated as below:

$$T_e = G(v) + (T_0 - G(v))e^{-B(t_e - t_0)} \tag{10}$$

where

$$G(v) = \frac{A(v)}{B}. \tag{11}$$

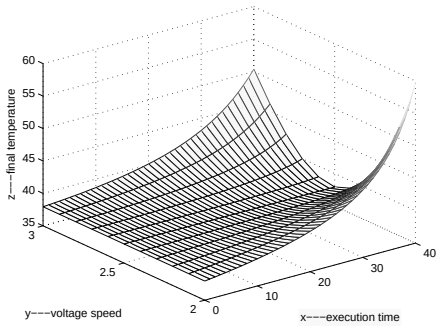


Fig. 2. Temperature varies with different supply voltages.

C. Motivating Example

We are not sure if there exist some general guidelines or we have to develop appropriate scheduling techniques case by case to minimize the peak temperature when the leakage/temperature relationship is taken into considerations. Therefore, we conducted some experiments to obtain some intuitions. We generated a large number of different schedules for a real-time job with deadline $D = 50$ and total workload as $E = 125$. We simulated the maximal temperature for each schedule and the results are shown in Figure 2.

From Figure 2, we can see that the peak temperatures by different schedules exhibit a "U" shape, and the peak temperature reaches its minimum when the lowest constant speed is applied. This seems to imply that using the lowest constant speed can minimize the maximal temperature. In what follows, we formulate this conclusion into a theorem and formally prove its correctness.

III. Minimize Peak Temperature

The experiments conducted in previous section seem to indicate that executing a real-time job with the lowest constant speed minimizes the peak temperature. This observation is valid under certain conditions. We formulate the conclusion by Theorem 1.

Theorem 1: Given a real-time job set $\mathcal{J}$, its deadline D and total execution time E, assume that the processor speed is continuously changeable. Then using the lowest constant speed that meets the deadline, i.e., $v_0 = E/D$, is the optimal scheduling solution in terms of minimizing the maximal temperature, if the following condition hold:

- $B > 0$;
- $G(v)$ is a non-negative, monotonically increasing, and convex function of v,

where B, G are defined in equation (9) and (11), respectively.

Proof Sketch: Due to page limit, we only prove the case that, for the two schedules S_1 and S_2 shown in Figure 1, the temperature by S_1 at $t = D$ is no greater than that by S_2. For simplicity, we set $D = 1$ and also assume that $T_{amb} = 0$.

Let $T(S_1)$ and $T(S_2)$ be the ending temperatures for S_1 and S_2, respectively. Then from equation (10), we have

$$\begin{aligned} T(S_1) &= G(v_0)(1-e^{-B}), \\ T(S_2) &= G(v_2)(1-e^{-B(1-x)}) + G(v_1)(1-e^{-Bx})e^{-B(1-x)}. \end{aligned}$$

To prove that $T(S_1) \le T(S_2)$, we only need to show that

$$\begin{aligned} G(v_0)(1-e^{-B}) &\le G(v_2)(1-e^{-B(1-x)}) \\ &+ G(v_1)(1-e^{-Bx})e^{-B(1-x)}, \end{aligned} \tag{12}$$

Or

$$G(v_0) \le kG(v_1) + (1-k)G(v_2), \tag{13}$$

where

$$k = \frac{e^{-B(1-x)} - e^{-B}}{1-e^{-B}}, 1-k = \frac{1-e^{-B(1-x)}}{1-e^{-B}}. \tag{14}$$

Since

$$v_0 = v_1 x + v_2(1-x), \tag{15}$$

and G_i is a convex function, we have

$$G(v_0) \le xG(v_1) + (1-x)G(v_2). \tag{16}$$

Therefore, to show that equation (13) holds, we only need to show that

$$xG(v_1) + (1-x)G(v_2) \le kG(v_1) + (1-k)G(v_2), \tag{17}$$

or

$$(G(v_1) - G(v_2))(x-k) \le 0. \tag{18}$$

As G_i is monotonically increasing and $v_1 < v_2$, so we have $G(v_1) \le G(v_2)$, and thus we only need to prove that

$$x \ge k = 1 - \frac{1-e^{-B(1-x)}}{1-e^{-B}}. \tag{19}$$

Or, equivalently,

$$\frac{1-e^{-B(1-x)}}{1-e^{-B}} \ge 1-x. \tag{20}$$

Now consider function

$$F(z) = \frac{1-e^{-Bz}}{1-e^{-B}} - z. \tag{21}$$

with $0 \le z \le 1$. We can readily show that function $F(z)$ is a concave function since $F''(z) < 0$. Note that the curve $F(z)$ passes two points, i.e. $(0,0)$ and $(1,0)$, as $F(0) = 0$ and $F(1) = 0$. Let $H(z)$ be the line that crosses these two points. Since $F(z)$ is concave, we have $F(z) \ge H(z) = 0$ for $0 \le z \le 1$. Therefore,

$$F(1-x) = \frac{1-e^{-B(1-x)}}{1-e^{-B}} - (1-x) \ge 0. \tag{22}$$

As a result, we prove that equation (19) and thus equation (19) holds. □

A. Justifications for Theorem 1

Theorem 1 holds only when several conditions are satisfied. In this subsection, we justify these conditions.

Consider equation (5). Note that $c_0 v(t)$ represents the leakage power at the ambient temperature, and $c_1 T(t)$ represents the increased leakage power consumption as temperature rises above the ambient temperature. From equation (4), it is not difficult to see that the leakage current increases as the temperature increases. Therefore, constants c_0 and c_1 must be non-negative and thus $A(v(t)) > 0$.

Moreover, based on (7), if $B = b - ac_1 < 0$, we would have

$$\frac{dT(t)}{dt} = A(v(t)) - BT(t) > 0, \tag{23}$$

and temperature will continue to increase indefinitely. This occurs only when the processor heat generation surpasses its heat removal capability, and thus the temperature will continue to rise and eventually cause the processor to break down. This scenario is called the "thermal run-away" [4]. Processor with this characteristic cannot work stably. Therefore, to avoid this scenario, $B > 0$ must hold. As a result, we can also conclude that

$$G(v) = \frac{A(v)}{B} \ge 0. \tag{24}$$

Theorem 1 also requires that $G(v)$ is a convex function of v. However, it is difficult to analytically prove that $G(v)$ is a convex function, since the temperature invariants c_0 and c_1 depend not only on the supply voltages but also on the technology parameters. Furthermore, c_0 and c_1 are obtained through curve-fitting rather than a closed analytical formula. In what follows, we try to make the justification empirically.

We built our processor model based on the work by Liao et al. [4] using the 65nm technology. We used (4) to compute the leakage currents for temperature from 40^oC to 110^oC with step size of 10^oC, and supply voltage from 0.65Volt to 1.05Volt with step size of 0.05V. These results were used to determine the temperature invariants c_0 and c_1 in (5) through curve-fitting. To obtain the leakage power consumption, the

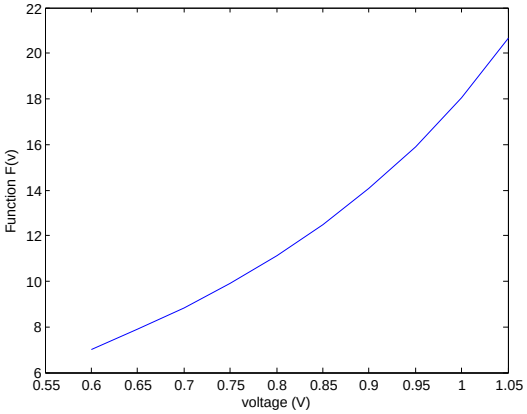


Fig. 3. Function $G(v)$ based on 65nm technology.

total number of gate, i.e., N_{gate} in (3), was set to be 10^6. The dynamic power consumption (and thus constant c_2) was determined based on the experimental results reported in [4] on benchmark *gcc*. For the thermal constants, we selected $Rth = 0.8K/W$, $Cth = 340J/K$ [2], and the ambient temperature was set to 25^oC. Figure 3 depicts the behavior of function $G(v)$ based on our experimental set up. We can clearly see from Figure 3 that function $G(v)$ is a non-negative, monotonic increasing, and most importantly, convex function. This justifies the conditions in Theorem 1.

IV. Conclusions and Future Work

As semiconductor technology continues to scale down in size, the positive feedback loop between the temperature and leakage becomes a critical issue not only for the power/energy minimization problem, but also for the temperature constrained design problem. In this paper, we intend to explore some fundamental principles that can be used when considering the leakage/temperature dependency in thermal aware real-time analysis. Our experimental results reveal that using the lowest constant speed is the optimal method to reduce the maximal temperature. We formulate this observation into a theorem and prove it formally. We also use empiric results to justify the conditions we present in the theorem. The significance of our work is that it clearly demonstrates the feasibility to incorporate the leakage/temperature into a more rigorous and analytical system level analysis. It also reveals a fundamental principle which can be applied in analyzing and developing leakage-aware temperature-constrained real-time scheduling techniques.

Our work can be extended in a number of ways. First, in this paper, we develop our theorem based on a processor model with continuously supply voltage. We want to extend this principle for processor models with discrete level of supply voltages. Second, this paper uses a very simple real-time model. How to extend the real-time model to a more practical and complex ones, such as those with priority assignments and preemption effects, will be an interesting problem. Note that, while our theorem seems to be very close to the well-known principles in power-aware scheduling, it does not mean that the existing methods for reducing energy consumption can be readily migrated for maximal temperature constraint. How to develop more effective techniques based on the principle we formulate in this paper will be an important future work for us. Third, our theorem is based on 65nm technology. As technology continues to scale down, it is not clear if all conditions supporting our theorem will still hold. Our next task is to study these cases.

Acknowledgment

This work is supported in part by NSF under Grants CNS-0545913, CNS-0917021, and CNS-0746643.

References

[1] ITRS, *International Technology Roadmap for Semiconductors (2005 Edition)*. Austin, TX.: International SEMATECH, http://public.itrs.net/.

[2] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," *ICSA*, pp. 2–13, 2003.

[3] L.-T. Yeh and R. C. Chu, *Thermal Management of Microelectronic Equipment: Heat Transfer Theory, Analysis Methods, and Design Practices*. New York, NY: ASME Press, 2002.

[4] W. Liao, L. He, and K. Lepak, "Temperature and supply voltage aware performance and power modeling at microarchitecture level," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 1042 – 1053, 2005.

[5] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: a temperature-aware model of subthreshold and gate leakage for architects," *University of Virginia Dept. of Computer Science Technical Report*, 2003.

[6] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware computer systems: opportunities and challenges," *IEEE Micro*, vol. 23, no. 6, pp. 52–61, 2003.

[7] N. Bansal, T. Kimbrel, and K. Pruhs, "Speed scaling to manage energy and temperature," *Journal of the ACM*, vol. 54, no. 1, pp. 1–39, 2007.

[8] G. Quan, Y. Zhang, W. Wiles, and P. Pei, "Guaranteed scheduling for repetitive hard real-time tasks under the maximal temperature constraint," *ISSS+CODES*, 2008.

[9] S. Zhang and K. S. Chatha, "Approximation algorithm for the temperature-aware scheduling problem," in *ICCAD*, 2007, pp. 281–288.

[10] R. Jejurikar, C. Pereira, and R. Gupta, "Dynamic slack reclamation with procrastination scheduling in real-time embedded systems," *DAC*, 2005.

[11] G. Quan and L. Niu, "Fixed priority scheduling for reducing overall energy on variable voltage processors," *RTSS'04*, Dec 2004.

[12] Y. Liu, H. Yang, R. P. Dick, H. Wang, and L. Shang, "Thermal vs energy optimization for dvfs-enabled processors in embedded systems," in *ISQED*, 2007, pp. 204–209.

[13] K. Skadron, T. Abdelzaher, and M. R. Stan, "Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management," in *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002, p. 17.

[14] M. Bao, A. Andrei, P. Eles, and Z. Peng, "On-line thermal aware dynamic voltage scaling for energy optimization with frequency/temperature dependency consideration," in *Design Automation Conference*, 2009, pp. 490–495.

[15] J.-J. Chen, S. Wang, and L. Thiele, "Proactive speed scheduling for real-time tasks under thermal constraints," *RTAS*, vol. 0, pp. 141–150, 2009.

[16] N. Fisher, J.-J. Chen, S. Wang, and L. Thiele, "Thermal-aware global real-time scheduling on multicore systems," *RTAS*, vol. 0, pp. 131–140, 2009.

[17] J. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 2003.

Leakage-Aware Scheduling for Real-Time Embedded Systems with QoS Guarantee

Linwei Niu
Department of Math and Computer Science
Claflin University
Orangeburg, SC 29115
linwei.niu@claflin.edu

***Abstract*—In this paper, we propose a dynamic approach to reduce both the dynamic and leakage energy consumption for real-time systems while ensuring the *Quality of Service (QoS)* guarantee. The QoS requirements are deterministically quantified with the *window-constraints*, which require that at least m out of each non-overlapped window of k consecutive jobs of a task meet their deadlines. Necessary and sufficient conditions for checking the feasibility of task sets with arbitrary service times and periods are developed to ensure that the *window-constraints* can be guaranteed in the worst case. And efficient scheduling techniques are proposed to merge the idle intervals. Through extensive simulations, our experiment results demonstrate that the proposed techniques significantly outperformed previous research in both overall and idle energy reduction.**

I. Introduction

Power aware computing has come to be recognized as a critical enabling technology in the design of pervasive real-time embedded systems. Many dynamic voltage scaling (DVS) based real-time scheduling techniques have been proposed to reduce energy. However, the energy saving achievable via voltage scaling is becoming severely limited with the dramatic increase of the leakage power consumption. To obtain the maximal reduction in the overall energy consumption, some researchers have also proposed more advanced DVS scheduling techniques, *e.g.* [1], [2], to reduce dynamic power and leakage power simultaneously. Most of them have targeted the hard real-time systems.

In recent years, there has been increasing interest that incorporates DVS scheduling techniques to deal with the power/energy conservation with regard to *Quality of Service (QoS)* guarantee. Two well known deterministic QoS guarantee models are the (m,k)-model [3] and the *window-constrained* model [4]. The (m,k)-model, proposed by Hamdaoui *et al.* [3], requires that m jobs out of any *sliding* window of k consecutive jobs of the task meet their deadlines, whereas the *window-constrained* model requires that m jobs out of each *fixed* and *nonoverlapped* window of k consecutive jobs meet their deadlines. The (m,k)-model adopts very strong constraints which may over-specify the QoS requirements for some applications. For example, some packet streams over the network can be logically divided into segments, and in each segment a minimum number of jobs must be completed in time. In this case, the (m,k)-model imposes a minimum requirement of job completions on each and every overlapped window of k jobs, regardless whether these jobs in different windows are logically related or not, which can be over stronger than necessary. The *window-constrained* model, on the other hand, is more appropriate for such kind of applications. In [5], Mok *et al.* proved that the general *window-constraint* scheduling problem for arbitrary service time and request period is NP-hard in the strong sense. So far, deterministic assurance with this model can only be guaranteed for very limited range of systems, such as those with all tasks having the same request periods and unit size execution times [5]. Note that the scheduling problem with $(m,k)-$guarantee is also NP-hard in the strong sense [6]. To guarantee the (m,k)-constraints, Ramanathan *et al.* [7] proposed to partition the jobs into *mandatory* and *optional* jobs. The mandatory jobs are jobs that must meet their deadlines in order to satisfy the (m,k)-constraints, while the optional jobs can be executed to further improve the QoS or simply be dropped. Inspired by that, we will also adopt the same strategy, *i.e.*, to partition the jobs into *mandatory* and *optional* jobs, in dealing with the problem of scheduling with *window-constraints* guarantee for task sets with arbitrary execution times and periods.

In this paper, we study the problem of reducing both the dynamic and leakage energy consumption for real-time systems with *window-constraints* guarantee. We propose an online algorithm to partition the jobs into mandatory/optional jobs dynamically according to the runtime environment such that the *window-constraints* can be guaranteed and no redundant jobs need to be executed. Moreover, scheduling techniques combining DVS and processor shut-down strategy are proposed to minimize the overall energy consumption.

The rest of the paper is organized as follows. Section II introduces the system models and some preliminaries. Section III introduces our pattern variation strategy and algorithm to reduce energy. The effectiveness of our approach is demonstrated using simulation results in section IV. In section V, we offer the conclusions.

II. Preliminaries

A. System model

The real-time system considered contains n independent periodic tasks, $\mathcal{T} = \{\tau_0, \tau_1, \cdots, \tau_{n-1}\}$, scheduled according to the earliest deadline first (EDF) policy. Each task contains an infinite sequence of periodically arriving instances called *jobs*. Task τ_i is characterized using five parameters, *i.e.*, (T_i, D_i, C_i, m_i, k_i). T_i, $D_i (D_i \leq T_i)$, and C_i represent the period, the deadline and the worst case execution time for τ_i, respectively. The *window-constraint* requirement for τ_i is represented by a

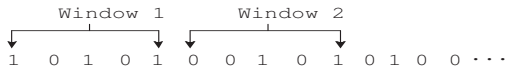


Fig. 1. The job patterns satisfy the $(m,k)-$constrait $(3,7)$ but fails to satisfy the *window-constraint* $(3/5)$ in Window 2.

pair of integers, *i.e.*, (m_i/k_i) $(0 < m_i \leq k_i)$, which require that, within each *non-overlapped* window of k_i jobs of τ_i, at least m_i jobs meet their deadlines. The j^{th} job of task τ_i is represented with J_{ij} and its arrival time, actual execution time and absolute deadline is represented by r_{ij}, C_{ij} and d_{ij}.

The system contains a DVS processor which can be in one of the three states: the *active*, *idle* and *sleeping* states. When the processor is idle, the major portion of the power consumption comes from the leakage which increases rapidly with the dramatic increasing of the leakage power consumption. Shutting-down strategy, *i.e.*, put the processor into its sleeping state, can greatly reduce the energy consumption when the processor is idle. However, it has to pay extra energy and timing overhead to shut down and later wake up the processor. Assuming that the power consumptions of a processor in its idle state and sleeping state are P_{idle} and P_{sleep}, respectively, the energy overhead of shutdown/wakeup is E_o, and the timing overhead is t_o. Then, the processor can be shut down with positive energy gains only when the length of the idle interval is larger than $T_{th} = \max(\frac{E_o}{P_{idle}-P_{sleep}}, t_o)$. We call T_{th} as the *shut down threshold interval*.

When the processor is active, the speed that can minimize the overall active energy for tasks is called the *critical speed* [1] and denoted as s_{crit}. Suppose the lowest possible speed for job J_k to finish by its deadline is s_k, then the optimal speed to execute job J_k should be $s_k^* = \max\{s_{crit}, s_k\}$.

B. Motivations

In [4], West *et al.* tried to establish the corresponding relationship between the *window-constrained* model and the $(m,k)-$model. They found that *window-constrained* model can be transferred into $(m,k)-$model, which can be summarized as follows: given a *window-constraint* (m_i/k_i), its corresponding $(m,k)-$contraint is $(m_i, 2k_i - m_i)$. However, the other transferring direction is not true, *i.e.*, given job patterns that can satisfy the $(m,k)-$contraint $(m_i, 2k_i - m_i)$, they do not necessarily satisfy the *window-constraint* (m_i/k_i), which is shown by the example in Figure 1. In Figure 1, the job patterns (in which "1" represents the mandatory job and "0" represents the optional job) can satisfy the $(m,k)-$constraint of $(3,7)$ in sliding windows, however, they cannot satisfy the *window-constraint* $(3/5)$ because in the second separate window, *i.e.*, "Window 2", there are only 2 mandatory jobs out of 5 jobs. To find the exact equivalence relationship between *window-constraint* and $(m,k)-$contraint for the general case is a challenging problem and needs further theoretical exploration. However, one closely tight sufficient condition to transfer $(m,k)-$contraint into *window-constraint* can be stated as follows:

Lemma 1: If the mandatory jobs for task τ_i can just satisfy the (m,k) constraint (m_i,k_i), then it can also satisfy the *window-constraint* of (m_i/k_i) and there is *no* redundant job.

Lemma 1 shows that the $(m,k)-$constraint (m_i,k_i) can directly be transferred into the *window-constraint* of (m_i/k_i). Since the $(m,k)-$constraint is stronger than the *window-constraint*, then one intuitive thought is that if a task set is

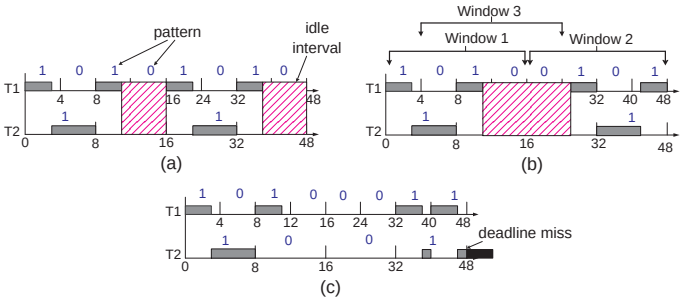


Fig. 2. (a) The task set $(\tau_1 = (4,4,3,2,4);\ \tau_2 = (8,8,5,1,2))$ schedule with E-pattern satisfies the $(m,k)-$constraints; (b) After pattern variation, the same task set schedule does not satisfy the $(m,k)-$constraints but still satisfies the *window constraints*; (c) The task set failed to be schedulable under arbitrary *window constrained* pattern;

schedulable under the $(m,k)-$pattern, it is also schedulable under any *window-constraint* pattern of the same window size. However, this does not necessarily hold. As shown in Figure 2, the task set $\{\tau_1, \tau_2\}$ is schedulable under $(m,k)-$constraints with the patterns defined in Figure 2(a), but is NOT schedulable with the patterns defined in Figure 2(c), although the patterns in Figure 2(c) can satisfy the *window-constraints*.

As shown, to reduce the overall energy consumption for real time systems with *window-constraints* guarantee, one has to deal with two highly co-related problems: (1) How to determine if a job should be mandatory or optional, and (2) How to schedule these jobs most efficiently. Two special *statically* defined mandatory/optional job partitioning strategies (patterns) were reported in literature. The first one is called *the deeply-red pattern* or R-pattern, which defines the pattern π_{ij} for a job J_{ij}, *i.e.*, the j^{th} job of task τ_i, by

$$\pi_{ij} = \begin{cases} 1 & \text{if } 0 \leq j \bmod k_i < m_i \\ 0 & \text{otherwise} \end{cases} \qquad j = 0,1,2,\cdots \tag{1}$$

The second one is called *the evenly distributed pattern* or E-pattern [7], which defines the pattern π_{ij} for a job J_{ij}, by

$$\pi_{ij} = \begin{cases} 1 & \text{if } j = \lfloor \lceil \frac{j \times m_i}{k_i} \rceil \times \frac{k_i}{m_i} \rfloor \\ 0 & \text{otherwise} \end{cases} \qquad j = 0,1,2,\cdots \tag{2}$$

The mandatory/optional job partitions according to equation (2) has the interesting property that it spreads out the mandatory jobs *evenly* in each task along the time. Moreover, it has the following interesting properties:

Lemma 2: Let the mandatory jobs for task τ_i be determined by equation (2). Then for any subsequence of task τ_i, the number of consecutive mandatory jobs is no more than $\lceil \frac{m_i}{k_i - m_i} \rceil$.

Although E-pattern has very good schedulability [8], it also has the problem that it will generate a lot of scattered idle intervals during which the leakage or the processor shutdown energy overhead will be considerable. For example, in Figure 2(a), the task schedule according to E-pattern generates two idle intervals within the first two windows. However, after careful pattern adjustment as shown in Figure 2(b), the idle intervals are merged into one and all mandatory jobs can still meet their deadlines. One interesting thing is that in the schedule in Figure 2(b) the $(m,k)-$constraint of task τ_1 is violated in Window 3 due to the strictness of (m,k) requirement, but the *window-constraints* are still satisfied in two separate windows Window 1 and Window 2, and thus all the time. Note that the pattern variation cannot be done arbitrarily as it will possibly make the resulting mandatory job set unschedulable. For example, the varied patterns in Figure 2(c) still satisfy the

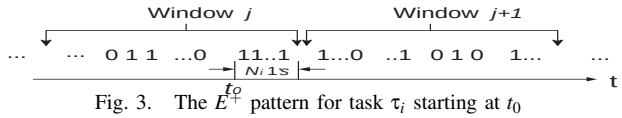


Fig. 3. The E^+ pattern for task τ_i starting at t_0

window-constraints, but the deadline missing on mandatory jobs of task τ_2 can not be avoided.

III. Dynamic pattern variation

As shown by the motivation example in Figure 2(b), dynamically varying the pattern can merge the idle intervals effectively. However, variation cannot be done arbitrarily. Otherwise, even the *window-constraints* can be satisfied, the schedulability of the mandatory job set in the resulting patterns cannot be guaranteed, as shown in Figure 2(c). Then the problem is: how to vary the pattern dynamically to merge the idle intervals without causing any deadline missing on mandatory jobs while still ensuring the *window-constraints*?

A. Pattern rotation – a guaranteed approach

To solve the problem above, some static analysis needs to be performed first. Due to the excellent schedulability of E-pattern, our pattern variation strategy is based on E-pattern. Moreover, close-form feasibility analysis based on E-pattern is provided in [8], which is very useful for us to set up the guarantee criteria. To explore how to vary the mandatory/optional job patterns based on E-pattern, we first introduce the following definitions:

Definition 1: The *Dynamic Sub-Window* of τ_i (denoted as $DW(k_i')$) is defined as the sub-window of the current window from τ_i (denoted as $DW(k_i)$) such that it contains only the rest k_i' patterns in $DW(k_i)$ starting from current time t to the end of $DW(k_i)$.

Definition 2: The *Critical Mandatory(or Optional) Point* of τ_i at time t (denoted as $CMT_i(t,DW(k_i'))$ for mandatory or $COT_i(t,DW(k_i'))$ for optional) is defined as the time point t' such that at t' the number of consecutive mandatory (or optional) jobs arriving at or after t within the *Dynamic Sub-Window* $DW(k_i')$ is maximized.

With the above definitions, our pattern variation strategies can be summarized in the following **Pattern Rotation Policies**. Specifically, given the current time t,

- (*Policy I*) If the processor starts to idle at time t, for each task τ_i, rotate the patterns in its *Dynamic Sub-Window* $DW(k_i')$ such that $|t-COT_i(t,DW(k_i'))|$ is minimized. If the idle interval after rotation reaches the end of the current window $DW(k_i)$, also rotate the patterns in next window $\overrightarrow{DW}(k_i)$ following $DW(k_i)$ such that $|t-COT_i(t,\overrightarrow{DW}(k_i))|$ is minimized;
- (*Policy II*) If the idle period expired at time t, for each task τ_i, rotate the patterns in its *Dynamic Sub-Window* $DW(k_i')$ such that $|t-CMT_i(t,DW(k_i'))|$ is minimized. If the busy period after rotation reaches the end of the current window $DW(k_i)$, also rotate the patterns in next window $\overrightarrow{DW}(k_i)$ following $DW(k_i)$ such that $|t-CMT_i(t,\overrightarrow{DW}(k_i))|$ is minimized;
- (*Policy III*) If an optional job $J_{ij} \in \tau_i$ meets its deadline at time t, rotate the patterns of τ_i in its *Dynamic Sub-Window* $DW(k_i')$ (in this case J_{ij} should also be included) such that $|r_{ij}-CMT_i(t,DW(k_i'))|=0$, while for each other task $\tau_j(\neq\tau_i)$, rotate the patterns in its *Dynamic Sub-Window* $DW(k_j')$ such that $|t-CMT_j(t,DW(k_j'))|$ is minimized.

The purposes of our pattern rotation policies are to make optional jobs and mandatory jobs from different tasks overlapped to the maximal extent such that the processor can be kept idle or busy for the longest time. Based on the pattern rotation policies, our online algorithm is presented in Algorithm 1.

B. E^+-pattern

It doesn't escape from our attention that the worst case of patten rotation is that: as shown in Figure 3, at time $t=t_0$, the maximal number of consecutive mandatory jobs within the *Dynamic Sub-Window* were move to the right end of the current window and might "interfere" with the mandatory jobs in the following window from other tasks. From Lemma 2, we know that there are at most $N_i=\lceil\frac{m_i}{k_i-m_i}\rceil$ consecutive mandatory jobs within the *Dynamic Sub-Window* of each task. The worst situation happens when, at time $t=t_0$, the N_i consecutive mandatory jobs from each task τ_i were rotated to the right end of its current window simultaneously. That will form a very interesting pattern phenomenon that, as shown in Figure 3, starting from t_0, there are N_i consecutive "1"s followed by the E-pattern for each task. To be convenient, we call it E^+-pattern. Without loss of generality, the E^+-pattern can also be defined statically as followed: let the pattern π_{ij} for job J_{ij}, *i.e.*, the j^{th} job of task τ_i, be defined by

$$\pi_{ij}=\begin{cases}1 & \text{if } j<N_i \text{ or } (j-N_i)=\lfloor\lceil\frac{(j-N_i)\times m_i}{k_i}\rceil\times\frac{k_i}{m_i}\rfloor\\ 0 & \text{otherwise}\end{cases}\qquad j=0,1,2,\cdots \tag{3}$$

For a mandatory job set determined with E^+-pattern, to check its schedulability, we have the following theorem:

Algorithm 1 The online algorithm. (Algorithm *LCDW*)

```
1:  Upon job completion:
2:  if the processor is idle then
3:     Rotate patterns for each task according to Policy I;
4:     t_cur = the current time;
5:     LST = the earliest arrival time for the upcoming mandatory
       jobs;
6:     if LST − t_cur > T_th then
7:        Shut down the processor and set up the wake up timer to
          be (LST − t_cur);
8:     else if the optional job queue Q_p is not empty then
9:        Select and run J_i ∈ Q_p with s_crit non-preemptively;
10:       if J_i is completed by its deadline then
11:          Rotate patterns for each task according to Policy III;
12:       end if
13:    end if
14: end if
15:
16:  Upon expiration of timer:
17: Rotate patterns for each task according to Policy II;
```

Theorem 1: Given system $\mathcal{T}=\{\tau_0,\tau_1,...,\tau_{n-1}\}$, let $\mathcal{E}^+$ be the mandatory job set according to their E^+-patterns, L be either the ending point of the first busy period or the *least common multiple* of T_i, whichever is smaller. Let $N_i=\lceil\frac{m_i}{k_i-m_i}\rceil$ and $W_i(0,t)$ represent the total mandatory work demand from task τ_i that arrive at or after time 0 and with deadlines less than or equal to t, *i.e.*

$$W_i(0,t)=\begin{cases}(1+\lfloor\frac{t-D_i}{T_i}\rfloor)C_i, & \text{if } t<N_iT_i\\ (\mathrm{N_i}+\lceil\frac{m_i}{k_i}(1+\lfloor\frac{t-N_iT_i-D_i}{T_i}\rfloor)\rceil)C_i, & Othewise\end{cases} \tag{4}$$

Then all mandatory jobs in $\mathcal{E}^+$ can meet their deadlines *iff*

$$\sum_i W_i(0,t)\le t \tag{5}$$

for all $t\le L$.

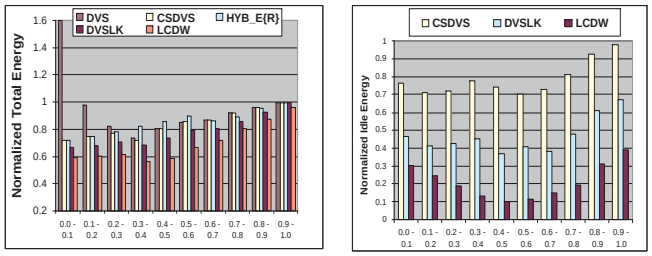


Fig. 4. (a) The total energy comparison; (b) The idle energy comparison.

From theorem 1, to check the feasibility of the $\mathcal{E}^+$ mandatory job set, we only need to check condition (5) on all deadlines of the mandatory jobs within the first busy period. And it guarantees that the *window-constraints* could be satisfied dynamically with our pattern rotation policies. Moreover, during the execution of the mandatory jobs, more advanced techniques can be adopted to reduce the energy further (will be discussed in a full version of this paper).

IV. Experimental results

For fair comparison, we only compare the energy performance of our approach with the approaches that can guarantee the *window-constraints* for *arbitrary* execution times and periods. For the first approach, the task sets are statically partitioned with E-pattern, and the mandatory jobs are executed with the highest processor speed. We refer this approach as *NoDVS* and use its results as the reference results. For the second approach *DVS*, the speeds of the tasks are scaled down as low as possible. For the third approach, the speeds of the tasks are scaled down to $s_k^* = \max\{s_{crit}, s_k\}$. We call this approach *CSDVS*. The fourth approach HYB_{E^R} is the hybrid approach in [8] but with processor speeds scaled down to no less than the critical speed s_{crit}. For the fifth approach *DVSLK*, the mandatory job set and the processor speeds are determined in the same way as *CSDVS*. But when the processor is idle, the upcoming mandatory jobs are delayed with the approach in [2] (We didn't compare with $CS-DVS-P$ in [1] because its procrastination algorithm based on utilization is not applicable when *window-constraints* are imposed). The sixth approach, namely *LCDW*, is our approach introduced in Section III. The processor model used in our experiments is the same as that in [1]. The periodic task sets tested were generated with the periods and the worst case execution times randomly chosen in the range of $[10ms, 50ms]$ and $[1ms, 10ms]$, respectively. The deadlines of the tasks were set to be less than or equal to their periods. The m_i and k_i for the *window-constraints* were also randomly generated such that k_i is uniformly distributed between 2 to 10, and $m_i < k_i$. The energy consumption for each approach was normalized to that by *NoDVS*, and the results are shown in Figure 4(a) and (b).

From Figure 4(a), voltage scaling without considering leakage will cause dramatic increase in the total energy consumption. For example, when the utilization is between [0.0, 0.1], the energy consumption by *DVS* is more than 60% of that by *NoDVS*. On the contrary, voltage scaling with leakage in mind can reduce the total energy consumption significantly. Procrastinating the mandatory jobs without pattern adjustment can help reduce the total energy further. For example, compared with *CSDVS*, *DVSLK* can reduce the total energy by up to 12%. However, with our pattern rotation and delay strategy, the energy reduction can be achieved more significantly. For example, the energy reduction by *LCDW* over *CSDVS* can be up to 28% and that by *LCDW* over *DVSLK* can be up to 20%. Also note that although HYB_{E^R} can reduce the dynamic energy efficiently as shown in [8], its performance degrade severely when both dynamic and leakage energy are considered. For example, when the utilization is between [0.2, 0.6], the total energy consumption by HYB_{E^R} is even higher than that by *CSDVS*. This is mainly because HYB_{E^R} optionally executed quite a few redundant jobs whose energy cost cannot be compensated by the savings in dynamic energy alone. Obviously pattern variation without leakage in mind becomes insufficient in terms of overall energy reduction.

With the scaling of IC technology and the dramatic increasing of the leakage, the energy consumption during the processor idle times will soon become a significant part of the total energy consumption. Figure 4(b) shows the average idle energy consumptions by the different approaches. Note that, our approach, *i.e.*, *LCDW* can always lead to much better idle energy savings than the previous approaches. As seen in Figure 4(b), *LCDW* can save around 80% idle energy compared with *CSDVS* and more than 50% idle energy compared with *DVSLK*. This is because, by rotating the pattern adaptively and delaying the mandatory jobs, *LCDW* can merge the idle intervals more aggressively than the other approaches. As a result, the processor can stay in the sleeping mode for much longer time.

V. Conclusions

In this paper, we propose a dynamic approach to minimize the overall energy consumption for real-time systems while guaranteeing the given QoS requirement in terms of *window-constraints*. Different from the previous work which can only guarantee the *window-constraints* for tasks with the same periods and unit size execution times, our approach can support task sets with arbitrary service times and periods. Moreover, we present efficient scheduling techniques to reduce the overall energy by varying the job patterns and procrastinating the execution of mandatory jobs and thus to merge the idle intervals. The effectiveness of our approach was demonstrated by the experimental results.

References

[1] R. Jejurikar, C. Pereira, and R. Gupta, "Leakage aware dynamic voltage scaling for real-time embedded systems," *DAC*, 2004.

[2] L. Niu and G. Quan, "Reducing both dynamic and leakage energy consumption for hard real-time systems," *CASES'04*, Sep 2004.

[3] M. Hamdaoui and P. Ramanathan, "A dynamic priority assignment technique for streams with (m,k)-firm deadlines," *IEEE Transactions on Computes*, vol. 44, pp. 1443–1451, Dec 1995.

[4] R. West, Y. Zhang, K. Schwan, and C. Poellabauer, "Dynamic window-constrained scheduling of real-time streams in media servers," *IEEE Trans. on Computers*, vol. 53, no. 6, pp. 744–759, June 2004.

[5] A.K.Mok and W.Wang, "Window-constraint real-time periodic task schedulling," *RTSS*, 2001.

[6] G. Quan and X. Hu, "Enhanced fixed-priority scheduling with (m,k)-firm guarantee," in *RTSS*, 2000, pp. 79–88.

[7] P. Ramanathan, "Overload management in real-time control applications using (m,k)-firm guarantee," *IEEE Trans. on Paral. and Dist. Sys.*, vol. 10, no. 6, pp. 549–559, Jun 1999.

[8] L. Niu and G. Quan, "Energy minimization for real-time systems with (m,k)-guarantee," *IEEE Trans. on VLSI, Special Section on Hardware/Software Codesign and System Synthesis*, pp. 717–729, July 2006.

An Adaptive Approach to Reduce Control Delay Variations

Shengyan Hong and Xiaobo Sharon Hu
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
{shong3,shu}@nd.edu

M.D. Lemmon
Department of Electrical Engineering
University of Notre Dame
Notre Dame, IN 46556
lemmon@nd.edu

***Abstract*—For many control systems, control performance is strongly dependent on delay variations of the control tasks. Such variations can come from a number of sources including task preemptions, variations in task workloads and perturbations in the physical environment. Existing work has considered improving control task delay variations due to task preemption only. This paper presents a general adaptive framework that incorporates a powerful heuristic aiming to further reduce delay variations. Preliminary results indicate that the heuristic significantly improves existing approaches.**

I. Introduction

For many cyber-physical systems, intelligent coordination between control design and its corresponding computer implementation can lead to improved control performance and/or reduced resource demands [1], [11], [14]. A prime example that benefits from such coordination is regulating delay variations (jitter) in control tasks. For many control systems, control performance strongly depends on delay variations in control tasks. Such variations can come from numerous sources including task preemptions, variations in task workloads and perturbations in the physical environment, and can cause degraded control system performance, such as sluggish response and erroneous behavior.

There are a number of published papers related to reducing delay variations. In [2], [5], the authors proposed a task decomposition based approach where each task is partitioned into three subtasks, i.e., Initial, Mandatory, and Final Subtasks (referred as the IMF model), and the delay variation of the final subtask (corresponding to control update) is minimized. A somewhat indirect way of reducing delay variations is to reduce task deadlines, which has been investigated by many researchers, e.g., [3], [4], [7], [12]. A common theme of all these methods is to focus on reducing deadlines of either tasks or subtasks. Because deadlines are only allowed to be reduced, these methods cannot effectively explore the design space where deadlines of certain tasks/subtasks may be increased (within some upper bounds) to reduce the overall delay variations.

The task-decomposition based methods [2], [5] suffer less, but still obvious performance degradation (compared with direct deadline reduction methods) when deadlines are only allowed to be decreased greedily. The decomposition task model is acceptable for control tasks where only a small amount of data needs to be passed to control update subtasks, otherwise context switching cost could be prohibitive. In addition, these methods require repeated worst-case response time computation under EDF, which can be time consuming and unsuitable for on-line use. On-line adjustment is needed to reduce delay variations when parameters such as task periods change in response to environment perturbations.

In this paper, we propose an on-line adaptive approach which directly minimizes delay variations for both decomposable and non-decomposable control tasks simultaneously. The approach leverages the IMF based task model for both types of tasks and formulates the delay variation minimization problem as an optimization problem. An efficient algorithm is designed based on the generalized elastic scheduling heuristic [10]. The efficiency of the algorithm readily supports an adaptive framework which can adjust deadlines of control tasks on-line in response to dynamic changes in workloads.

II. Preliminaries

In this section, we first introduce necessary notation and scheduling properties and then present some motivation for the problem to be solved.

A. System Model

We consider a computer system which needs to handle a set Γ of N real-time control tasks, $\{\tau_1, \tau_2, \cdots, \tau_N\}$, each with the following attributes: (C_i, D_i, P_i), where C_i is the worst case execution time (WCET) of τ_i, D_i is τ_i's deadline, P_i is its period, and $C_i \leq D_i \leq P_i$. Without loss of generality, we adopt the IMF task modeling approach introduced in [5]. Specifically, we let τ_i be composed of three subtasks, the initial part τ_{ii} for sampling input data, the mandatory part τ_{im} for executing the control algorithm, and the final part τ_{if} to deliver the control action. Thus, a task set Γ_{IMF} consists of $3N$ subtasks $(\tau_{1i}, \tau_{1m}, \tau_{1f}, ..., \tau_{Ni}, \tau_{Nm}, \tau_{Nf})$, each with the following parameters

$$\tau_{ii} = \{C_{ii}, D_{ii}, P_i, O_{ii}\}$$
$$\tau_{im} = \{C_{im}, D_{im}, P_i, O_{im}\}$$
$$\tau_{if} = \{C_{if}, D_{if}, P_i, O_{if}\}$$

where $O_{i\star}$ is the offset of the corresponding subtask. Note that in order for the IMF model to faithfully represent the original task set, each τ_{ii} must be executed before τ_{im}, which must in turn be executed before τ_{if}. For a non-decomposable task, say τ_i, we simply have $C_{ii} = C_{im} = D_{ii} = D_{im} = 0$, and $C_{if} = C_i$. Some tasks may also be partially decomposable, i.e., we may have non-zero C_{ii} and D_{ii} but $C_{im} = D_{im} = 0$.

TABLE I
A MOTIVATIONAL EXAMPLE CONTAINING FOUR TASKS WITH THE FIRST TWO TASKS BEING INDECOMPOSABLE.

Task name	Computation Exec. time	Deadline	Period	Original Delay Variations (%)	New Delay Variations (%) DRB / TBB / ADVR	Delay Variations before Reassignment (%) DRB / TBB / ADVR	Delay Variations after Reassignment (%) DRB / TBB / ADVR
Speed	5000	27000	27000	18.52	38.98 / 18.52 / 3.7	38.98 / 18.52 / 3.7	41.48 / 18.52 / 3.7
Strength	8000	30000	320000	1.56	2.89 / 1.56 / 3.37	28.91 / 15.63 / 33.68	31.25 / 15.63 / 22.81
Position	10000	45000	50000	32	31.75 / 26 / 0.27	31.75 / 26 / 0.27	34 / 26 / 2.34
Sense	13000	60000	70000	40	32.86 / 20 / 8.57	32.86 / 20 / 8.57	32.86 / 20 / 8.57

To achieve desirable control performance, control actions should be delivered at regular time intervals periodically. However, preemptions, variations in task workloads, and perturbations in the physical environment make each instance of the control actions experience different delays. Similar to [5], we define the delay variation as the difference between the worst and best case response times of the same final subtask relative to its period, i.e.,

$$DV_i = \frac{WCRT_{if} - BCRT_{if}}{P_i}, \tag{1}$$

where $WCRT_{if}$, $BCRT_{if}$ are the worst case response time and best case response time, respectively. The definition of delay variation gives information on the delay variance that a task will suffer in the control action delivery within a period. Our problem then is to minimize the delay variations of all the final subtasks.

We use Earliest Deadline First (EDF) scheduling algorithm. A necessary and sufficient condition for a synchronous task set to be schedulable under EDF is given below.

Theorem 1. *A set of synchronous periodic tasks with relative deadlines less than or equal to periods can be scheduled by EDF if and only if* $\forall L \in K \cdot P_i + D_i \leq \min(L_{ip}, H, B_p)$ *the following constraint is satisfied,*

$$L \geq \sum_{i=1}^{N} (\lfloor \frac{L - D_i}{P_i} \rfloor + 1) \cdot C_i \tag{2}$$

where $L_{ip} = \frac{\sum_{i=1}^{N}(P_i - D_i)U_i}{1-U}, U_i = \frac{C_i}{P_i}, U = \sum_{i=1}^{N} \frac{C_i}{P_i}$, $K \in N$*(the set of natural numbers including 0),* H *is the hyperperiod, and* B_p *is the busy period [6], [9].*

For an asynchronous task set, the condition in Theorem 1 can be used as a sufficient condition [6].

B. Motivation

We use a simple robotic example, similar to the one in [5], to illustrate the deficiencies of existing approaches for delay variation reduction. The example contains four control tasks, i.e., the speed, strength, position and sense tasks. The tasks and the original delay variations under EDF are shown in columns 1 to 5 of Table I. We consider two representative methods for delay variation reduction: a deadline reduction based method from [4], denoted as DRB, and a task decomposition based method from [5], denoted as TBB.

Suppose that decomposing the speed and strength tasks would cause non-negligible context switch overhead and we opt to only partition the position and sense tasks according to the IMF model. Assume that the IMF decomposition is made considering that the initial and final subtasks consume 10% of the execution time of corresponding control task. By applying the DRB and TBB methods, new delay variation values are obtained and are shown as the first two values in column 6 of Table I. It is easy to observe that the TBB method is more effective in reducing delay variations than the DRB method (which does not even provide much improvement over the original delay variations). However, with the TBB method, two tasks still suffer about 20% or more delay variation.

Now, assume that at some time interval, the execution rate of the strength task increases by 10 times. If the same deadline assignments are used for the tasks/subtasks, the delay variation of the strength task increases to 28.91% and 15.63% for DRB and TBB, respectively (see the first two values in column 7 of Table I). Suppose we apply the DRB and TBB methods online in response to the period change, the new delay variation values are shown in column 8 of Table I. It turns out that the TBB does not improve the delay variations while DRB actually gives worse delay variations.

With our proposed approach (ADVR), better delay variations can be obtained for all the cases considered above. In particular, for each respective scenario, we have applied our approach and the delay variation values are shown as the third number in columns 6-8 of Table I. Though for some tasks, delay variations see a small increase, most of the tasks which suffer from large delay variations due to the other methods are now having much smaller delay variations.

III. OUR APPROACH

From the previous section, one can see that delay variations could be improved significantly if more appropriate deadline assignments can be identified. In this section, we describe our proposed adaptive delay variation reduction (ADVR) approach. ADVR is built on three basic elements. First, the general IMF model as given in the last section is used for both decomposable and non-decomposable tasks. Second, the delay variation reduction problem is formulated as an optimization problem. Third, an efficient heuristic is developed to solve the optimization problem. The heuristic is then incorporated into a simple adaptive framework.

We adopt the generalized IMF task model described in Section II to represent the task set under consideration. The general IMF model allows both decomposable and non-decomposable tasks to be treated equivalently. Given an IMF

task set, there may exist numerous sets of feasible deadlines (D_{ii}, D_{im}, D_{if}) which allow the original task set to be schedulable. However, different sets of deadlines could lead to different delay variations of the original tasks. To find the particular subtask deadline assignment that results in the minimum delay variation, we formulate the deadline selection problem as a constrained optimization problem. Though existing work such as [10], [13] has considered the deadline selection problem as an optimization problem, there are two major differences between our present formulation and theirs. First, our formulation directly minimizes delay variations. Second and more importantly, our formulation leverages special properties of the IMF task model and thus allows much more effective delay variation reduction.

The delay variation minimization problem is to minimize the total delay variation bounds of (1) subject to the schedulability constraints as given in (2) while considering the IMF task model. Specifically, we have

$$\text{min: } \sum_{i=1}^{N} w_i (D_{if} - C_{if})^2 \tag{3}$$

$$\text{s.t. } \sum_{i=1}^{N} [(\lfloor \frac{L - D_{ii}}{P_i} \rfloor + 1) \cdot C_{ii} + (\lfloor \frac{L - D_{im}}{P_i} \rfloor + 1) \cdot C_{im}$$

$$+ (\lfloor \frac{L - D_{if}}{P_i} \rfloor + 1) \cdot C_{if}] \leq L, \forall L \in K \cdot P_i + D_i \leq L_{ip}, \tag{4}$$

$$D_{ii} = D_{im}, \tag{5}$$

$$C_{if} \leq D_{if} \leq \min(D_{im}, D_i - D_{im}), \tag{6}$$

$$C_{im} \leq D_{im} \leq D_i - C_{if}, \tag{7}$$

where L_{ip} and K is defined in Theorem 1. If task τ_i is not decomposable, (6) and (7) are replaced by

$$C_{if} \leq D_{if} \leq D_i, \tag{8}$$

$$D_{im} = 0. \tag{9}$$

To see why the above formulation can lead to valid deadline assignments that minimize delay variations, first note that deadline D_{if} is the upper bound of the WCRT of the final subtask of τ_i, and C_{if} is the lower bound of the BCRT of the final subtask of τ_i. By setting $w_i = \frac{1}{P_i^2}$, the objective function in (3) is the upper bound on the delay variation squares as defined in (1). (By using w_i instead of $\frac{1}{P_i^2}$ directly, we can also capture the relative importance of control tasks in the objective function.)

To guarantee schedulability under the IMF model, we have introduced a set of constraints in our formulation. Constraint (4) helps ensure the schedulability of the task set according to Theorem 1. However, this constraint alone is not sufficient since there exist dependencies when executing the initial, mandatory and final subtasks of any decomposable task. Note that ensuring the subtask dependencies during task execution is straightforward. The difficulty lies in capturing this in the schedulability test without being overly pessimistic.

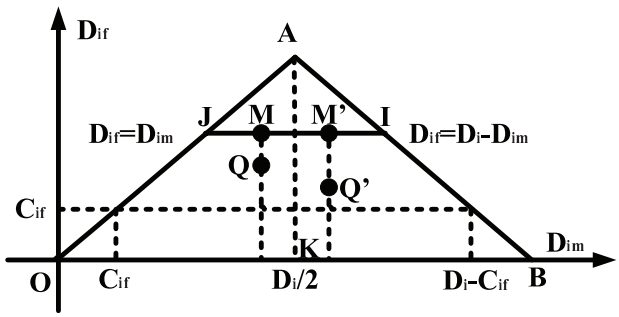


Fig. 1. Feasible deadline region for mandatory/final tasks.

To handle the unique challenges due to the IMF task model, we have added several more constraints in addition to (4). To capture the fact that τ_{ii} is always executed before τ_{im}, we can set $D_{ii} \leq D_{im}$ (and hence τ_{ii} has a higher priority than τ_{im} as long as $O_{ii} = O_{im} = 0$). For simplicity, we let $D_{ii} = D_{im}$, assuming that a tiebreak goes to τ_{ii}, which is constraint (5).

Since τ_{if} must start after τ_{im} is completed, we let $O_{if} = D_{im}$. Furthermore, to guarantee that task τ_i finishes by its deadline D_i, we must have $O_{if} + D_{if} \leq D_i$. We thus have $D_{if} \leq D_i - D_{im}$, which leads to one part of (6). The other part of (6), i.e., $D_{if} \leq D_{im}$ reflects the desire that smaller deadlines should be assigned to the final subtask compared to that of the mandatory subtask so as to help the delay variation reduction of the final subtask (as this would be the delay variation of interests). (7) constrains the space of D_{im} and is obtained simply by combining $D_{im} \leq D_i - D_{if}$ and $D_{if} \geq C_{if}$.

Based on constraints (5)-(7), Figure 1 depicts the feasible region of (D_{im}, D_{if}), which is bounded by $\triangle ABO$. To make our search more efficient, we would like to reduce our search region as much as possible without sacrificing the optimization solution quality. Theorem 2 provides the basis for reducing the search region.

Theorem 2. *Given a set Γ_{IMF} of N tasks. If the necessary and sufficient condition for schedulability in Theorem 1 is satisfied for a synchronous task set Γ_{IMF} with $(D_{ii} = D_{im}, D_{im}, D_{if})$ for $i = 1, ..., N$, then the same condition is satisfied for a synchronous task set Γ'_{IMF} with $(D'_{ii}, D'_{im}, D_{if})$, where $D'_{ii} = D'_{im} \geq D_{im}$ for $i = 1, ..., N$.*

Applying Theorem 2 to the search region depicted in Figure 1, one can readily see that point M' on the segment MI leads to a schedulable solution if point M leads to a schedulable solution. Since D_{im} corresponding to M' is larger than that of M, M' is a more desirable solution than M as it leads to a smaller D_{if}. Based on this observation, we can reduce the search region by 1/2 by replacing constraints (6)-(7) in the optimization problem by the following:

$$C_{if} \leq D_{if} \leq D_i - D_{im}, \tag{10}$$

$$\frac{D_i}{2} \leq D_{im} \leq D_i - C_{if}, \tag{11}$$

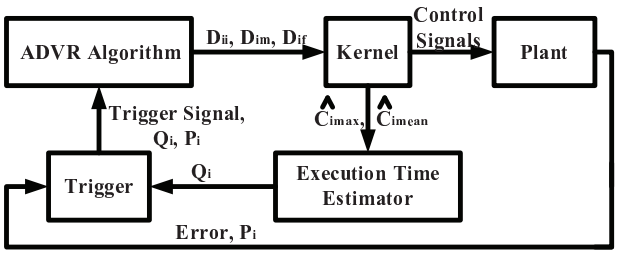


Fig. 2. Adaptive framework for delay variation reduction.

If task τ_i is not decomposable, constraint (11) is replaced by

$$D_{im} = 0. \tag{12}$$

Solving the optimization problem specified in (3) together with (4), (5), (10), (11) and (12) is not trivial as it involves dealing with a discontinuous function (the floor function). Heuristic techniques such as the one presented in [10] can be used to solve the problem, but it will take many iterations to reach convergence. Besides, the simplified sufficient condition adopted by [10] either fails to find a solution or finds a very pessimistic solution for task sets with high utilization. We are developing a better heuristic to avoid such problems. Our heuristic first replaces constraint (4) by two related constraints

$$\sum_{i=1}^{N}[(\lfloor\frac{L-D_{ii}}{P_i}\rfloor+1)\cdot C_{ii}+(\lfloor\frac{L-D_{im}}{P_i}\rfloor+1)\cdot C_{im}$$
$$+(\lfloor\frac{L-D_{if}}{P_i}\rfloor+1)\cdot C_{if}]\leq L, \forall L\in K\cdot P_i+D_i<L_{ip}, \tag{13}$$

$$\sum_{i=1}^{N}[(\frac{L-D_{ii}}{P_i}+1)\cdot C_{ii}+(\frac{L-D_{im}}{P_i}+1)\cdot C_{im}$$
$$+(\frac{L-D_{if}}{P_i}+1)\cdot C_{if}]\leq L, \forall L=L_{ip}, \tag{14}$$

where L_{ip} and K are as defined in Theorem 1. It is easy to see that (4) is equivalent to (13) and (14). We are still working on devising our whole improved heuristic by making use of some observations specific to the particular problem at hand and also by designing a more effective search method. We refer to this heuristic as ADVR.

As we have seen from the motivational example, dynamic workload changes could cause larger delay variations if the original task/subtask deadlines were used. It is desirable to deploy an on-line adaptive framework to adjust task/subtask deadlines when workloads change significantly. The key to such an adaptive framework is an efficient method of solving the optimization problem posed earlier. Based on preliminary results, our heuristic, ADVR, seems to satisfy such a requirement. Hence, we propose an adaptive framework built on ADVR. Our proposed framework is similar to the one in [8] and is shown in Figure 2. In this framework, an on-line monitoring mechanism in Kernel measures the mean execution time $\hat{c}_i$ and the maximum execution time $\hat{C}_i$, and sends these measured data to Execution Time Estimator. Execution Time Estimator computes the current execution time estimate Q_i and forwards it to Trigger. Meanwhile, Plant also reports its error, i.e. the difference between the actual and ideal performances of Plant, and task period P_i to Trigger. When the error and the changes of Q_i and P_i reach some thresholds, Trigger will signal ADVR algorithm to recompute the deadlines and send the results to Kernel. With these new results, Kernel adjusts Plant so as to reduce delay variations.

IV. Summary and Future Work

We have presented a new approach to reduce delay variations of control tasks. The approach formulates the delay variation reduction problem as an optimization problem that can effectively handles both decomposable and non-decomposable tasks. Based on several key observations, we are devising an efficient heuristic to solve the optimization problem. The efficiency of the heuristic will lead to an adaptive framework that can dynamically readjust task/subtask deadlines to keep delay variations small in the presence of environment perturbations. As future work, we will implement and evaluate our approach in a real-time operating system to control an actual application. We will measure the effect of our algorithm on the control performance of the application under various physical perturbations. We will also test and improve the convergence speed of our algorithm.

V. Acknowledgement

This work is supported in part by NSF under grant numbers CNS07-20457 and CNS09-31195.

References

[1] P. Albertos, A. Crespo, I. Ripoll, M. Valles, and P. Balbastre, "Rt control scheduling to reduce control performance degrading," in *ICDC '00*.
[2] P. Balbastre, I. Ripoll, and A. Crespo, "Control tasks delay reduction under static and dynamic scheduling policies," in *RTCSA '00*.
[3] ——, "Optimal deadline assignment for periodic real-time tasks in dynamic priority systems," in *ECRTS '06*.
[4] ——, "Minimum deadline calculation for periodic real-time tasks in dynamic priority systems," *IEEE Trans. Comput.*, vol. 57, no. 1, pp. 96–109, 2008.
[5] P. Balbastre, I. Ripoll, J. Vidal, and A. Crespo, "A task model to reduce control delays," *Real-Time Syst.*, vol. 27, no. 3, 2004.
[6] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Syst.*, vol. 2, no. 4, 1990.
[7] E. Bini and G. Buttazzo, "The space of edf deadlines: the exact region and a convex approximation," *Real-Time Syst.*, vol. 41, no. 1, pp. 27–51, 2009.
[8] G. Buttazzo and L. Abeni, "Adaptive workload management through elastic scheduling," *Real-Time Syst.*, vol. 23, no. 1/2, pp. 7–24, 2002.
[9] G. C. Buttazzo, "Hard real-time computing systems: Predictable scheduling algorithms and applications." Springer, 2005.
[10] T. T. Chantem, X. S. Hu, and M. D. Lemmon, "Generalized elastic scheduling for real-time tasks," *IEEE Trans. Comput.*, vol. 58, no. 4, pp. 480–495, 2009.
[11] A. Crespo, I. Ripoll, and P. Albertos, "Reducing delays in rt control: The control action interval, decision and control," in *IFAC '99*.
[12] H. Hoang, G. Buttazzo, M. Jonsson, and S. Karlsson, "Computing the minimum edf feasible deadline in periodic systems," in *RTCSA '06*.
[13] T. Kim, H. Shin, and N. Chang, "Deadline assignment to reduce output jitter of real-time tasks," in *IFAC '00*.
[14] M. Lluesma, A. Cervin, P. Balbastre, I. Ripoll, and A. Crespo, "Jitter evaluation of real-time control systems," in *RTCSA '06*.

Towards Timing Decomposition for Scalable Robot Control: Collision Detection Analysis

Hoon Sung Chwa, Jinkyu Lee and Insik Shin
Dept. of Computer Science
KAIST, South Korea
chwahs@cps.kaist.ac.kr, jinkyu@cps.kaist.ac.kr, insik.shin@cs.kaist.ac.kr

***Abstract*—Scalable control over a large number of robots raises many challenges. Several programming languages have been proposed to support scalable robot programming paradigms, where users can program the behavior of entire robots as a whole, rather than the behavior of individual robots. Unfortunately, such paradigms have little support on timing properties. For example, users may want to specify real-time behavior of a group of robots (i.e., moving all robots to the same destination in 10 seconds), hoping that such a system-level timing constraint can be decomposed into the timing behavior of individual robots (i.e., move a robot to the destination in 5 seconds and move another robot to the same destination in 10 seconds, in order to avoid a collision between these two robots). Our primary goal is to develop a timing decomposition framework that can decompose the system-level path & velocity planning on a group of real-time robots into local path & velocity planning of individual robots, preserving the real-time requirements imposed to the system. Achieving such a goal involves many problems to address. Typically, collision is one of the major obstacles to the development of such a timing decomposition. This paper presents our work in progress. We first present techniques to analyze collision detection in an efficient manner. The proposed analysis technique employs the notion of warning region, and this notion helps to reduce search space from a 3-D coordinate system to a 2-D coordinate system. Building upon this analysis, we then consider velocity planning with power optimization. Specifically, we want to determine the moving speed of individual robots such that they can reach a destination within a global deadline while minimizing the total power consumption.**

I. Introduction

There has been a growing attention to scalable control over a very large number of robots, such as swarm robot [1] and modular robotics [2]. Several studies (*e.g.*, [3], [4]) have been introduced for supporting scalable robot programming. For instance, the Meld language [3] enables users to program many robots as if the users control a single robot. However, these studies did not yet pay much attention to accommodating timing properties in a scalable manner. It lacks a systematic methodology that allows users to specify a system-level (global) timing property of entire robots without having to concern how to transform the system-level timing properties into the local timing properties of all individual robots.

We aim at the development of scalable control methodology that takes into account timing constraints. As an example, let us consider an emergency evacuation problem as follows. Suppose there are a large number of robots in a room, where there is a single exit, and an emergency (*i.e.*, fire) takes place in the room. A user may want all the robots to evacuate out of the room to be safe. The user can give to entire robots a global operation such as "all robots go through the exit in 30 seconds." Such a global operation needs to be decomposed into local operations that individual robots can perform. Then, the global operation needs to be transformed into the problem of determining the moving speed of individual robots such that all robots can move out of the room through the exit within the deadline of 30 seconds without any collision. Towards this, there are many problems to address. We need to address path planning to find a path towards a destination. We also need to resolve velocity planning to reach a destination within a deadline with no collision. And this entails collision detection analysis to see whether a collision will happen with a given configuration of path and velocity.

Our primary goal is to address the above problems so as to develop scalable control methods for real-time robots. In this paper, we just present the current work in progress; We introduce efficient techniques to analyze collision detection as a basis for scalable control techniques. Collision detection involves a large search space, which includes two-dimensional physical space and one-dimensional time. Our collision detection technique consists of two steps: one step for physical space and the other for time. Such decoupling inherently allows efficient way of detecting collisions, mostly reducing a 3-D search space to 2-D search space. In addition to efficiency, such decoupling also makes it easier to extend our techniques for more sophisticated configurations. Building upon such collision detection analysis techniques, we can then consider many velocity planning issues with a variety of optimization criteria, such minimization of the longest traveling time or minimization of total power consumption subject to deadlines. In this paper, we focus on velocity planning with power optimization. We are currently working on the measurement of power consumption of various DC motors to derive a trade-off between motor speed vs. power consumption. We are also working on the development of efficient way of finding optimal solutions to the power minimization problem.

The rest of this paper is organized as follows. Section II describes our system model. In Section III, we show how to analyze collision detection. In Section IV, we present velocity planning with power optimization based on collision detection

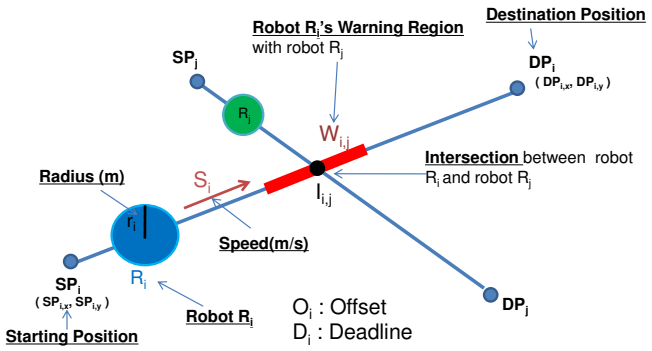


Figure 1. Notations

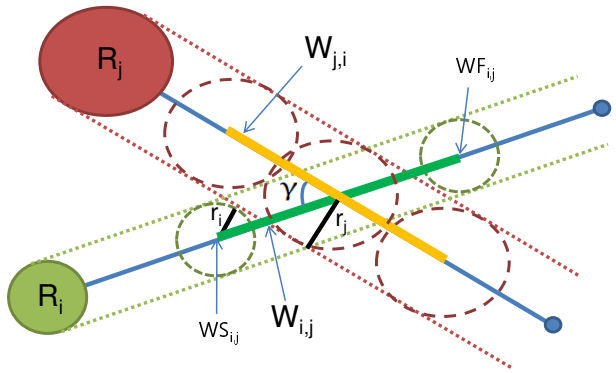


Figure 2. Warning region

analysis. Finally, we conclude our paper with future work in Section V.

II. System model

In this paper, we mainly consider collision detection and velocity planning for robots. Here, we present the assumptions and notations used throughout the paper (notations are also illustrated in Figure 1):

1) Each robot has starting (SP_i) and destination (DP_i) positions.
2) Each robot has a straight path and a constant fixed speed (S_i).
3) Each robot has an offset (O_i) which is the time to start.
4) The shape of each robot is circle with a different radius (r_i).
5) Each robot does not collide with the other robots when they are at starting positions.

We define the **path** function P_i of robot R_i to represent the mapping from robot's path to the global two-dimensional (2D) coordinate system, and it is described as

$$\mathbf{P_i} : l_i \in [0, L_i] \longmapsto P_i(l_i) = z \in \mathbf{Z} \tag{1}$$

where L_i is the total distance (Euclidean distance between SP_i and DP_i) of robot R_i and $z \in \mathbf{Z}$ is a point in 2D-workspace. When robot R_i goes a distance of l_i from a starting position, the robot is located at

$$z : (SP_{i,x} + l_i \cos\theta, SP_{i,y} + l_i \sin\theta)$$

where z is the point on 2D-workspace, and θ is the angle from x-axis to the path.

The **velocity profile** function indicates the relationship between robot traveling time and distance, and it is described as

$$\mathbf{V_i} : t \in [0, T_i] \longmapsto V_i(t) = l_i \in [0, L_i] \tag{2}$$

with $V_i(0) = 0$ and $V_i(T_i) = L_i$. The *total traveling time* of robot R_i refers to $T_i = \frac{L_i}{S_i} + O_i$. During t amount of time, robot R_i travels

$$l_i = \begin{cases} S_i \cdot (t - O_i) & \text{if } t - O_i > 0 \\ 0 & \text{otherwise.} \end{cases}$$

The **intersection** between path P_i and P_j is defined as

$$I_{i,j} = l_i \in [0, L_i] \; s.t. \; P_i(l_i) = P_j(l_j), \; l_j \in [0, L_j]. \tag{3}$$

Note that $I_{i,j}$ refers to robot R_i's distance that intersects with path P_j.

III. Collision Detection

In this section, we study how we resolve the collision detection problem. There are three factors that determine collisions among robots; radius, angle between two paths, and speed of robot. we divide the problem into two steps according to the factors. The first step (called warning region analysis) is connected with space, while the second step (called velocity analysis) is related with time. The first step computes a region where it is possible for two robots to collide, and it can be determined by two factors, which are radius and angle between two paths. The second step extends the first step by adding the speed factor. This step checks whether there is a collision or not by considering when robots enter and exit warning regions. Designing it as a two-step approach inherently allows us to reduce complexity and to enhance extensibility.

A. Step 1: Warning Region Analysis

For robots R_i and R_j, we define a **warning region** $W_{i,j}$, which is a set of all points on path P_i where it is possible to collide with robot R_j, described below:

$$W_{i,j} = \{l_i | \; \mid I_{i,j} - l_i \mid \; \leq \frac{r_i + r_j}{\cos(90 - \gamma)}\} \triangleq [WS_{i,j}, WF_{i,j}] \tag{4}$$

where r_i and r_j are radius of each robot, and γ (use vector inner product) is the angle between two paths as shown in Figure 2. We refer to $WS_{i,j}$ and $WF_{i,j}$ as the *entering* and *exiting* distance from a starting position of robot R_i.

The length of warning region is

$$|W_{i,j}| = |W_{j,i}| = 2\left(\frac{r_i + r_j}{\cos(90-\gamma)}\right). \quad (5)$$

B. Step 2: Velocity Analysis

Now, we add speed factor to the warning region (derived from Step 1). We define a **collision time interval** $C_{i,j}(V_i)$, which is a time interval when robot R_i and robot R_j collide with each other, described below:

$$\begin{aligned} C_{i,j}(V_i) &= V_i^{-1}(W_{i,j}) \\ &= [V_i^{-1}(WS_{i,j}), V_i^{-1}(WF_{i,j})] \end{aligned} \quad (6)$$

where V_i^{-1} is the inverse function of robot R_i's velocity profile and returns the time interval during which R_i passes the warning region $W_{i,j}$. Note that the *collision time interval* of robots (denoted as $C_{i,j}(V_i)$ and $C_{j,i}(V_j)$) can be different according to their speed. The robot R_i's total traveling time in the warning region $W_{i,j}$ is

$$|C_{i,j}(V_i)| = \frac{|W_{i,j}|}{S_i} \quad (7)$$

where $|W_{i,j}|$ is the length of warning region, and S_i is the speed of robot R_i.

Let $CS_{i,j}$ and $CF_{i,j}$ denote the *entering* and *exiting* time instants at which robot R_i enters and exits warning region $W_{i,j}$, and they are defined as

$$CS_{i,j} = V_i^{-1}(WS_{i,j}) \ , \ CF_{i,j} = V_i^{-1}(WF_{i,j}) \quad (8)$$

where $V_i^{-1}(WS_{i,j})$ and $V_i^{-1}(WF_{i,j})$ are described in Eq. (6). The collision time interval (denoted as $[CS_{i,j}, CF_{i,j}]$) is described as:

$$\begin{aligned} CS_{i,j} &= V_i^{-1}(I_{i,j}) - \frac{|C_{i,j}(V_i)|}{2}, \\ CF_{i,j} &= V_i^{-1}(I_{i,j}) + \frac{|C_{i,j}(V_i)|}{2} \end{aligned} \quad (9)$$

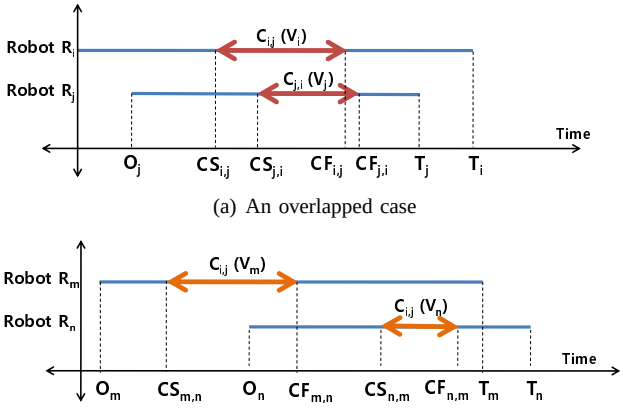
where $I_{i,j}$ is the intersection between path P_i and P_j, and $|C_{i,j}(V_i)|$ is the length of collision time interval.

With this collision time interval, we check whether the collision between robot R_i and robot R_j occur or not. As described in Figure 3, if two collision time intervals of each robot are overlapped (non-overlapped), two robots collide (do not collide) with each other.

In the non-overlapped case described in Figure 3(b), we find the following relation:

$$\begin{aligned} &CS_{m,n} > CF_{n,m} \text{ or } CS_{n,m} > CF_{m,n} \\ \iff &\text{ No collision between robot } R_m \text{ and } R_n. \end{aligned} \quad (10)$$

The following theorem offers conditions for collision detection.



(a) An overlapped case

(b) A non-overlapped case

Figure 3. Velocity analysis

***Theorem** 1:* Given the starting and destination positions, a speed, an offset, and a radius of each robot, there is no collision between robots if

$$CS_{i,j} > CF_{j,i} \text{ or } CS_{j,i} > CF_{i,j}, \text{ for } 1 \le i < j \le N,$$

where N is the number of robots.

Proof: Eq. (10) can be easily extended to the general case of any arbitrary number of robots. The theorem then immediately follows. ■

By defining warning region ($W_{i,j}$) and velocity analysis ($C_{i,j}(V_i)$), we can check the collision between two robots just comparing collision time interval ($[CS_{i,j}, CF_{i,j}]$) of them. When checking collision, we do not have to examine all robots for every time interval. It is sufficient to check the collision time interval of two robots.

C. Discussion

In this subsection, we first discuss how efficient our proposed collision detection analysis is. Through the two-step analysis, we can reduce search space from a 3-D coordinate system to a 2-D coordinate system. As shown in Figure 2, each robot has 3-D coordinate consisting of its position (x, y) and time t. Once we transform the original search space to what is shown as in Figure 3, we only need to control its time and one-dimensional position according to velocity (a 2-D coordinate system). In addition to reducing the dimension of coordinates, our analysis reduces search space even further. For the domain of time, a naive approach would seek for collision every time t. However, due to the velocity analysis, it is enough to check collisions only when robots enter and exit collision time intervals. Furthermore, when checking collisions, we do not have to consider all the other robots. Due to the warning region analysis, it is enough to consider only the robots that are involved in the same warning region. This way, our proposed analysis technique performs collision detection in

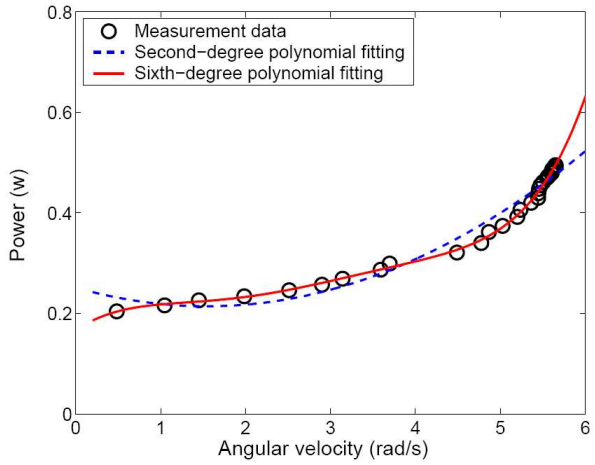


Figure 4. Power consumption according to velocity

a very efficient way, and it is suitable for scalable collision detection analysis.

Another direction of discussion is about how our assumptions can be relaxed. For clarity of presentation and space limit of this paper, we assume a straight path and a constant speed in Section II. When we relax these assumptions by considering more sophisticated paths or a variety of speeds, such relaxation would naturally require some extensions to either of the warning region analysis step or the velocity analysis step one by one, but not to both steps together. That is, when we make changes to the assumptions on path, the warning region analysis should be modified to accommodate such changes, but we can keep the same velocity analysis techniques. Similarly, variation of speed only affects the velocity analysis step. Therefore, we can relax our assumptions without having to change all the analysis steps, but some of the steps only.

IV. Velocity Planning with Power Optimization

With our collision detection algorithm, we can address many kinds of optimization problems by controlling the speed and offset of robots. One of the typical optimization problems is minimizing total power consumption. We consider *power optimization problem*: when there are multiple robots that have an initial configuration, such as starting and destination positions, and a deadline, and a power vs. speed trade-off, find each robot's speed and offset that minimize total power consumption.

We set up DC motors' power model and analyze the trade-off between power and speed. According to Figure 4 (we obtain this figure from [5]), we denote the trade-off between power and speed as $\rho(\nu)$. We formulate power optimization problem as

$$\text{minimize} \sum_{n=1}^{N} \rho(\nu_n) \times (T_n - O_n) \tag{11}$$

subject to :

$$CS_{n_1,n_2} > CF_{n_2,n_1} \text{ or } CS_{n_2,n_1} > CF_{n_1,n_2} \quad \text{for } 1 \leq n_1 < n_2 \leq N, \tag{12}$$

$$0 \leq T_n \leq D_n \tag{13}$$

where ν_n is robot R_n's speed, T_n is total traveling time, Eq. (13) is for collision detection, and T_n is less than equal to deadline (denoted as D_n).

V. Conclusion and Future Work

This paper presents collision detection analysis techniques. The proposed techniques allow to check collisions in an efficient manner. It employs two steps to decouple space and time dimensions, and such decoupling allows a considerable search space reduction in many aspects, including one from 3-D coordinates into 2-D coordinates. The proposed collision detection techniques serve as a basis for many other robot problems, such as velocity planning and path planning.

We are currently measuring a trade-off between speed and power consumption for DC motors and plan to derive an appropriate formula ($\rho(\nu_n)$) on it. Given such $\rho(\nu_n)$, our future work also includes developing efficient power optimization algorithms to the problem given in Section IV. If the formulation turns out to be a non-convex optimization problem, we consider developing heuristics to find sub-optimal solutions. We are also working on implementing a timing decomposition framework with many robots.

Acknowledgement

This research was supported in part by IT R&D program of MKE/KEIT of Korea [2009-KI002090, Development of Technology Base for Trustworthy Computing], National Research Foundation of Korea (2009-0086964), and KAIST ICC, KIDCS, KMCC, and OLEV grants.

References

[1] G. C. Pettinaro, I. W. Kwee, L. M. Gambardella, F. Mondada, D. Floreno, S. Nolfi, J. louis Deneubourg, and M. Dorigo, "Swarm robotics: A different approach to service robotics," in *Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS '02*, 2002.

[2] M. P. Ashley-Rollman, M. D. Rosa, S. S. Srinivasa, P. Pillai, S. C. Goldstein, and J. D. Campbell, "Declarative programming for modular robots," in *Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS '07*, 2007.

[3] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, "Meld: A declarative approach to programming ensembles," in *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007, pp. 2794–2800.

[4] M. D. Rosa, S. C. Goldstein, P. Lee, J. D. Campbell, and P. Pillai, "Programming modular robots with locally distributed predicates," in *Proceedings of the 2008 IEEE International Conference on Robotics and Automation*, 2008, pp. 3156–3162.

[5] Y. Mei, Y.-H. Lu, Y. C. Hu, and C. G. Lee, "Energy-efficient motion planning for mobile robots," in *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*, 2004, pp. 4344–4349.

Implementing Transactions in a Distributed Real-Time System without Global Time

A. Burns and Y. Chen
Department of Computer Science
University of York, UK
email: burns@cs.york.ac.uk

***Abstract*—A simple algorithm is presented for implementing and analysing real-time transactions executing on a distributed platform. The algorithm does not require global time, but does not suffer from excessive jitter.**

I. Introduction

In distributed real-time systems it is necessary to implement application code as *transactions* that incorporate processing elements on one or more processors and communications across one or more networks [3]. For distributed systems built upon a time-triggered architecture [2] the implementation and analysis of transactions is straightforward. With event-triggered architectures that do not require or support a global time base the implementation of transactions is not as simple if output jitter needs to be controlled. In this short paper we provide an implementation scheme and associated analysis for event-triggered systems. Note even when the basic architecture is event-triggered there will still be the need to support periodic tasks and therefore periodic transactions. We therefore assume the existence of local clocks on each node. Although there is no global time service, any two clocks will be assumed to have bounded drift.

To constraint the flow of work through the system some form of flow control is needed. But again this does not necessarily require global time. Also output jitter, at the end of the transaction, need to be bounded. Issues of composability require bounded behaviour at all nodes of the systems and it can be argues that end-to-end latency can be trade again composability [4] – this is not however discussed in this paper.

II. Standard Analysis with Global Time

Consider, as a means of illustrating the approach, a simple periodic transaction that has two processing parts τ_1 and τ_2 executing on different processors, and a communication link l. Task τ_1 inputs data from the environment, does some initial processing and then passes its 'result' to the link. Task τ_2 takes this results, undertakes further processing and produces an output for the environment. The transaction is simply: $\tau_1 \xrightarrow{l} \tau_2$. Task τ_1 is a pure periodic task that has a defined period T and has a simple structure such as the following:

```
Start_Time := clock
write Start_Time to link
next_release := Start_Time
loop
  input from environment
  undertake processing
  write result to link l
  next_release := next_release + T
  delay_until next_release
end loop
```

Using standard response time analysis it is possible to calculate R_1 the worst-case response time of this task. It will also be possible to estimate the worst-case transmission time for the link, R_l. The calculation of R_l will, of course, depend upon the network protocol. Note the values R_1 and R_l will be known prior to execution and will constitute common knowledge in the system.

In a time-triggered system any reading of a local clock is defined to give a global value. The code for τ_2 could therefore take the following form:

```
read Start_Time from link
next_release := Start_Time + R_1 + R_l
loop
  delay_until next_release
  read from link l
  undertake processing
  write result to the environment
  next_release := next_release + T
end loop
```

The advantage of this structure is that τ_2 is immediately executing with the right period and is guaranteed (within the bounds of the analysis) to have data available on the link when it executes the read operation. The read operation is non-blocking; if when a read is attempted no data is available then there is a fault that can, potentially, be dealt with.

The disadvantage comes from the need to support a global time base and the pessimism that arises from assuming that the communicated data can arrive as late $R_1 + R_l$ after the release of τ_1. Although both of these values are genuinely worst-case, it is not in general true that a transaction can suffer both worst-case situations at the same time. And hence there may be pessimism in the offset value used to separate the executions of τ_1 and τ_2.

When there is no global time base then all that is known at the second processor is the common knowledge of the period of the transactions. Any data arriving on the link may have been transmitted early in the cycle or towards the actual worst-case latency on transmission.

III. Analysis without Global Time

The simplest way to implement a transaction on a distributed platform without global time is to allow each task (apart from the first one) to execute the following simple loop.:

```
loop
  read from link
  undertake processing
  write to next link
end loop
```

Unfortunately this suffers from extreme output jitter. It is also difficult to derive an estimate of the worst-case behaviour as transactions can catch up with one another. Typically some form of *rate control* is applied to stop data been passed on 'too early'. In the following this idea is extended to produce a new protocol called NGT (No Global Time).

Assume that the read operation on the link has the following semantics. It blocks until data is available, and it returns, as well as the data, the earliest time the data was available to be read. So if the data is already available when the call of read is made, the operation succeeds immediately and the time returned is the time that the network interface placed the data in the appropriate buffer for the application code. If the data is not available the call is held until the data is communicated and the time returned is then the current clock value. The code for τ_2 is as follows (but note τ_1 no longer communicates its start time). The first task is assumes to start at time 0; in the following `t` is the time returned from the link (as defined above). *A global set of times is used to illustrate the behaviour of the protocol – but these values are not needed or the protocol to function.*

```
next_release := 0
read from link l returning t
loop
  undertake processing
  write result to the environment
                   (or next link)
  next_release := max(next_release,t) + T
  delay_until next_release
  read from link l returning t
end loop
```

So the arrival of the data sets up the period of the task. Initially, if the data arrives early in the cycle, the read operation will subsequently block and the effective 'period' of the task will be greater than T. But once the maximum latency for the data has been experienced τ_2 will behave as a purely periodic task will period T.

For example, assume time starts at 0, the period of the transaction is 20, the response times of τ_1 are initially 5, 7, 7, 6, 8, 5 and the transmission times of the resulting communication are 12, 13, 14, 14, 12, 12. Task τ_2 would behave as follows. Its first read operation would block until the data arrive at time 17 (5+12). It would then calculate its next release to be at time 37 (max(0,17)+20). Its second read would again block until time 40 (20+7+13). The delay time would now be 60 (max(37,40)+20).

The third data item arrives at time 61 (40+7+14) which will force the following loop to start at time 81. But now the worst case has been experienced. The 4th data message arrives at time 80 (60+8+12), so the read operation at time 81 now does not block and the task loops with a fixed period of 20 at times 101, 121 etc. (ie. an offset of 21). As long as the worst-case latency for the message has already been experienced the loop will now be purely periodic and all read operations will be non-blocking. Its start was however characterised by periods of 37, 23, 21, and 20 before this 20 value became fixed.

In terms of schedulability analysis, assuming a period of 20 is safe. The task will initially have a longer period, but this will not undermine any guarantee delivered by the schedulability test (as long as the test is sustainable[1] – which all standard tests are).

To complete an assessment of the example, note that the time triggered approach would require τ_2 to have an offset of 22. So it starts with a more regular execution but it has a longer latency in its normal phase.

A. Clock drift and infrequent worst-case behaviour

To cater for clock drift an occasional slightly shorter period can be added (i.e. a loop of 19). If this is too much the algorithm will force a 21 value on the subsequent iteration. Note if the clock drift is in the other direction (τ_2's processor clock running quicker) then the the algorithm will automatically extend one period by a small amount – a read operation will block.

If the worst-case message delay occurs very infrequently an application can decide to occasionally bring the period back from its maximum value (for example a one-off 19). This may result is a later period of 21 occurring. Overall jitter is increased but average (and normal) latency is reduced.

B. Fault recognition

One advantage of the time-triggered approach is that a fault (data not arriving) is immediately recognised. Without a global time service this is not as straightforward. There are however some bounds that can be derived. Once the algorithm has stabilised then all reads should be non-blocking so any delay can be interpreted as an error. However due to the reasons outlined above (e.g. clock drift) it would be necessary to give a tolerance on data arriving late.

A safe upper bound on a timeout value can be calculated as follows. With no other knowledge of actual execution and communication times the largest gap between two arrivals of the data is $2T$. Hence a timeout value of T is an upper bound (i.e. delay in the loop is T after the first arrival, and then wait up to T for the data to arrive). However an improvement on this can be obtained if one records how early data does arrive. If W is the maximum time that data has been in the input buffer waiting to be read then the timeout value can be reduced to $T - W$. The code would have the following form:

```
next_release := 0
read from link l returning t
W := 0
loop
  undertake processing
  write result to the environment
  next_release := max(next_release,t) + T
  delay_until next_release
  select
    read from link l returning t
  timeout T-W
    undertaken alternative action
  W := max(W, clock-t)
end loop
```

IV. Simulation Results

To evaluate the validity and performance of this NGT protocol a set of simulation experiments were undertaken. Here we report on one such experiment. The hardware platform was assumed to consist of ten nodes in a pipeline. A single repeating transaction runs through these nodes, with a single task per node. The period of the transaction (and hence the 'period' of each task) was 200ms. Each task had a maximum response time of 180ms, and each communication link had a maximum transmission time of 20ms. Actual response and communication times were obtained from a normal distribution constrained to have these maximum values.

Figure 1 illustrates the end-to-end latency values for the first 5,000,000ms of execution (ie. 25,000 executions of the transaction). Initially the latency is less than 1000ms, but this value grows until an interval of approximately 1800ms is obtained at the end of the simulation. For comparison it should be noted that the time-triggered protocol (TTP) would have a fixed latency of 1800ms plus the final task's execution. So an overall bound of 1980ms.

The NGT protocol produces a behaviour that approaches that of the time-trigger protocol (TTP). It corresponds to the worst-case actual behaviour of each task and communication link. If each worst-case behaviour can reach its theoretical limit then at that point NGT will give the same results as TTP. In the simulations the worst-case can be reached and so as the simulations continue the latency increases.

If the real upper bounds are below the 'worst-case' values used in the static analysis (as will often be the case), NGT will stabilise on a value below the theoretical worst-case. It reflects only the worst-case situations actually experienced by the system.

To reduce even this improved end-to-end latency, one can apply the 'recovery' techniques described earlier. In Figure 2 the period of each task is reduced from 200ms to 199ms every 10 invocations if the data was found to have arrived within this bound on each of these 10 invocations. Jitter is controlled by only making a change occasionally, but as a result the end-to-end latency is rarely above 1600ms.

V. Conclusion

A simple algorithm has been presented that allows a periodic transaction to dynamically set its own parameters on a distributed platform without a global time service. The tasks of the transaction, once each has experiences its maximum latency for its input data, will execute as regular periodic tasks with a fixed period. An advantage of the proposed scheme is that the maximum latency through the transaction is minimised. There is no need to set a potentially pessimistic offset for later components of the transaction. Rather the protocol learn how long each task must wait to get a smooth flow of data through the system.

The only requirement on the hardware platform is that data, as it arrives on an input link, must be time-stamped with the local time of arrival. This is a straightforward operation for a network interface card.

The proposed scheme can deal with clock drift and is able to respond to omission failures (of the input data). It is also able to bring back the worst-case behaviour in a controlled way. So, for example, if the communications media experience a glitch that pushed the end-to-end latency out to an excessive level then the protocol would, over time, bring this value back. But would do so in a way that had a small effect on output jitter.

References

[1] S.K. Baruah and A. Burns. Sustainable schedulability analysis. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 159–168, 2006.

[2] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings 16th IEEE Real-Time Systems Symposium*, 1995.

[3] J.P. Gutierrez, J.G. Garcia, and M. González Harbour. On the schedulability analysis for distributed real-time systems. In *proceedings 9th Euromicro Workshop on Real-Time Systems*, pages 136–143, 1997.

[4] S. Matic and T.A. Henzinger. Trading end-to-end latency for composability. In *RTSS*, pages 99–110. IEEE, 2005.

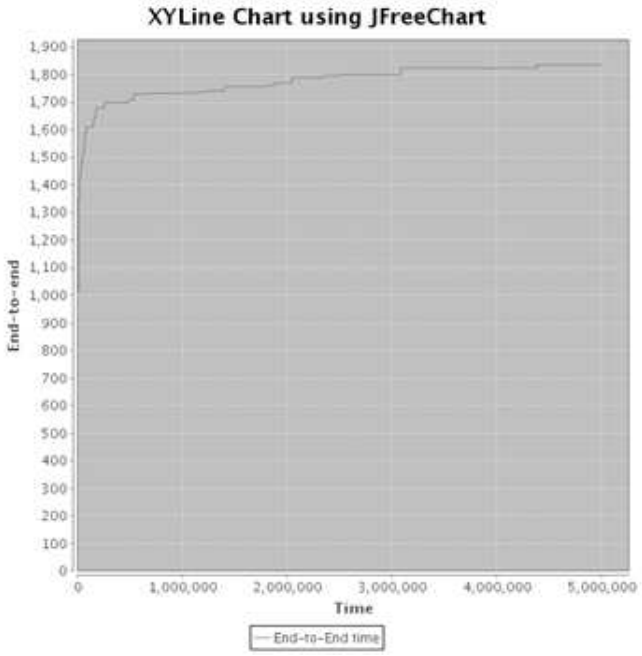


Fig. 1. No Recovery

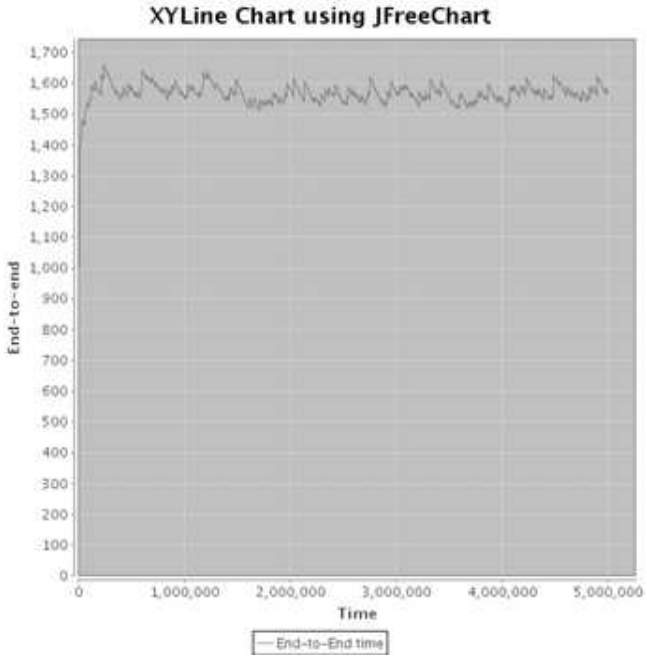


Fig. 2. With Recovery

Statistical-based Response-Time Analysis of Systems with Execution Dependencies between Tasks

Yue Lu, Thomas Nolte, Johan Kraft and Christer Norström
Mälardalen Real-Time Research Centre
Mälardalen University, Västerås, Sweden
{yue.lu, thomas.nolte, johan.kraft, christer.norstrom}@mdh.se

Abstract

This paper presents a novel statistical-based approach to Worst-Case Response-Time (WCRT) analysis of complex system models. These system models have been tailored to capture intricate execution dependencies between tasks, inspired by real industrial control systems. The proposed WCRT estimation algorithm is based on Extreme Value Theory (EVT) and produces both WCRT estimates together with a predictable probability of being exceeded (i.e., 10^{-9}*). By using the tools developed, we validate the proposed method by evaluating a model taken from the real industrial control system, and we show the results in comparison with other four analysis methods.*

1 Introduction

To date, most existing embedded real-time software systems have been developed in a traditional code-oriented manner, i.e., making extensive use of legacy software. Many such systems are maintained over extended periods of time, sometimes spanning decades, during which the systems become larger and increasingly complex. The result is that these systems are difficult and expensive to maintain and verify. There are many embedded systems existing in industry which consist of millions of lines of C code, corresponding to 50, or 100 tasks or more, where many tasks have real-time constraints. The example of such systems is the robotic control systems developed by ABB [1]. Looking closer at these systems, contrary to the assumption in most real-time theory, tasks exhibit strong temporal dependencies, e.g., asynchronous message-passing, globally shared state variables and runtime changeability of periods and priorities of tasks, which vary the execution time of the tasks radically.

One desirable approach to avoid the timing-related errors in such complex systems is to use schedulability analysis methods, such as Response-Time Analysis (RTA) [2]. Nevertheless, RTA (and other schedulability analysis techniques), although providing the prediction about timing behavior of execution in worst-case scenarios, rely on the existence of a fixed Worst-Case Execution-Time (WCET) of the tasks. Correspondingly, the quality of the analysis is directly correlated to the quality of the WCET estimates. Unfortunately, in the above described systems, the WCET of tasks obtained by static WCET analysis techniques may not easily be bounded. Sometimes a pessimistic WCET bound can be calculated based on maximum queue lengths. While in other cases the WCET is completely unbounded until the behavior of dependent tasks is known. Consider the following example in Figure 1, taken from an industrial robotic control system, where a task reads all messages buffered in a message queue and processes them accordingly:

```
1 msg = recvMessage(MyMessageQueue);
2   while (msg != NO_MESSAGE){
3   process_msg(msg);
4   msg = recvMessage(MyMessageQueue);
5 }
```

Figure 1. Iteration-loop wrt. message passing

By using static WCET analysis, the upper bound of number of messages actually consumed is equal to the maximum queue size. Furthermore, other tasks with a higher priority may preempt the execution of the loop and refill the queue at runtime. Looking further at the corresponding task periodicity dependencies, the analysis performed at RTA level also contributes to the pessimism as the number of loop iterations is not supposed to be bounded by the maximum queue size when preemption occurs.

The other approach, which avoids the state-space explosion issue raised by model checkers such as UPPAAL [3] and TIMES [4], for instance, is to use simulation-based methods that sample the state space. The first type of simulation technology to use is Monte Carlo simulation, which can be described as keeping the highest result from a set of

randomized simulations. Several frameworks already exist in this realm, such as the commercial tool *VirtualTime* [5] and the academic tool *ARTISST* [6]. However, the main drawback of using Monte Carlo simulation is the low state-space test coverage, which subsequently decreases the confidence in the results of finding rare worst-case scenarios. The other category is to apply an optimization algorithm (e.g., (meta)heuristic search algorithm), on top of Monte Carlo simulation, as in [7] and [8], which yield substantially better results, i.e., tighter lower bounds of the WCRT estimation.

Another approach is to use stochastic analysis of hybrid task sets in priority-driven soft real-time systems, as in [9]. Nevertheless, this approach does not allow for dependencies between tasks in the analysis, and the priority of jobs (a task is comprised by a sequence of jobs) and task periods are fixed.

In this paper, we present a novel statistical-based approach to response time analysis of systems with intricate execution dependencies between tasks. The proposed method uses samples collected by running Monte Carlo simulation as the input, and produces WCRT estimates on tasks along with a predictable probability of being exceeded, i.e., 10^{-9}.

2 Modeling of Complex Real-Time Systems

The system model used in this work describes the detailed execution dependencies between tasks with respect to resource usage and interaction, e.g., Inter-Process Communication (IPC), CPU execution time and logical resource usage. Practically, the model is specified by the modeling language used in RTSSim [10], which can be considered as a domain-specific language describing both architecture and behavior of task-oriented systems developed in C, and running on a single processor. Its syntax and semantics are as expressive as the C programming language, and include the typical RTOS services to the task models, such as task scheduling (e.g. Fixed-Priority Preemptive Scheduling), IPC via message passing and synchronization (semaphore). RTSSim employs a hierarchical model to specify the system structure consisting of a number of *tasks*. Each task is characterized by a *period*, a *constant offset*, a *maximum jitter*, and a *priority*. Periods and priorities can be changed at any time by any task in the application. Finally, each task is composed of a number of *jobs* and invoked RTOS services. The interested reader can refer to [10] for a thorough description of RTSSim.

3 Extreme Value Theory

Extreme Value Theory (EVT) [11] is a separate branch of statistics for dealing with the tail behavior of a distribution. It is used to model the risk of the extreme, rare events, without the vast amount of sample data required by a brute-force approach. The example applications are hydrology, material sciences, telecommunications etc.

There are three models in EVT, i.e., the Gumbel (type I), Frechét (type II) and Weibull distributions (type III), which are intended to model random variables that are the maximum or minimum of a large number of other random variables. It is worth noting that the Frechét distribution is bounded on the lower side ($x > 0$) and has a heavy upper tail, while the Weibull model relates to minima (i.e., the smallest extreme value). Since the purpose of this work is to find the higher response time of the tasks in rare worst-case scenarios, we therefore use the maximum case in the Gumbel distribution, referred to as the Gumbel Max in the reminder of the paper.

4 WCRT Estimation Based on EVT

The proposed method, *WCRTEVT* is shown in Algorithm 1. It is a recursive procedure which takes as argument m data sets, of which each contains N samples of the response time of the task under analysis. The algorithm returns the WCRT estimation with a predictable probability of being exceeded (i.e., 10^{-9}). It consists of the following two steps: 1) construction of the referenced data sets, 2) WCRT estimation of the referenced data sets using EVT.

4.1 The Referenced Data Sets

In order to construct the input data sets to the *WCRTEVT*, there are m Monte Carlo simulations in RTSSim to run at first. Then the n best simulations with the highest maximum value of response times, are selected as the referenced data sets. For each referenced data set, there are N (i.e., N is no less than 9 000) samples of the response time taken from the task under analysis. This sufficiently ensures making a good estimate. The construction is showed in rows 1-3 in Algorithm 1, where x_i in line 3 is the highest response time of the task under analysis observed in simulation per each data set.

4.2 WCRT Estimation of the Referenced Data Sets

4.2.1 Blocking of N Samples

In order to avoid the risk of mistakenly fitting raw response time data that may not be from random variables, to the Gumbel distribution, we use the method of block maxima [11], as proposed in [12]. This is done by grouping N response time samples in each referenced data set into k blocks of size is b, and then choosing the maximum

value from each block to construct a new set of sample "block maximum" values, i.e., $Y \leftarrow y_{i,1}, ..., y_{i,k}$, $y_{i,k} \leftarrow maxima(S) \leftarrow N_{(k-1)\times b+1}, ..., N_{kb}$ as shown in row 6, 9 and 10 in Algorithm 1. The samples at the end of the execution sequence in a simulation that do not completely fill a block are discarded. For instance, if there are 9 samples per data set, i.e., $\{1119, 1767, 2262, 2287, 1792, 2687, 1942, 1842, 1692\}$, and b (i.e., the size of the blocks) is 2, then the last sample (i.e., 1692) in the sequence is discarded since it can not be grouped in the 4 (i.e., $\left\lfloor \frac{9}{2} \right\rfloor$) blocks. Furthermore, the initial value of b is 100.

4.2.2 The Best-fit Gumbel Max Parameters Estimation

The estimation of the parameters of the Gumbel Max distribution is the core of *WCRTEVT*, which is also an iterative procedure as shown in rows 8-35 in Algorithm 1. The selection of b is a trade-off between the quality of fit to the Gumbel Max distribution, and the number of blocks (i.e., k) in each data set available used in the estimation of the Gumbel parameters. In this paper, we introduce two procedures using two different search algorithms, i.e., *lwbsearch* and *upbsearch* which could find the proper value of b producing the best-fit Gumbel Max parameters estimation. The algorithm *lwbsearch* is invoked at first as shown in rows 8-26 in Algorithm 1, which focuses on searching for the value of b to be as low as possible. In this way, there are more blocks, i.e., the bigger value of k, used as samples in the estimation. However, in some cases, *lwbsearch* may fail in finding such value of b in best-fit tests. If this is the case, then *upbsearch* will be adopted, which is showed in rows 27-35 in Algorithm 1. Moreover, the best-fit test is in terms of examining the estimated Gumbel parameters by using a goodness-of-fit (GOF) test, i.e., Chi-square test. Note that other more advanced (meta)heuristic search algorithms can be applied. While the empirical results including the one presented in Section 5 and the ones have not been included in this paper due to space limitations, show that the two proposed algorithms work well enough to reach the goal. There is one more interesting point to highlight, i.e., the generally accepted value of k is 30 as introduced in [12]. Therefore, in this work, the size of blocks b should be smaller than $\left\lfloor \frac{N}{30} \right\rfloor$. For the sake of space, we can not give the detailed explanation about each search algorithm, as well as their implementation.

4.2.3 The WCRT Estimations Formula

The two parameters of the Gumbel Max distribution: a location parameter μ and a scale parameter β, are used in the Gumbel percent-point function, which returns the WCRT estimation that the block maximum Y cannot exceed with a certain probability q, as shown in Equation 1.

$$est = \mu - \beta \times log(-log((1 - P_e)^b)) \quad (1)$$

Algorithm 1 $WCRTEVT(m)$

1: $RT \leftarrow rt_1, ..., rt_m \leftarrow MonteCarlo(m, rnd\_inst())$
2: $n \leftarrow \frac{m}{100}$
3: $X \leftarrow x_1, ..., x_i, ..., x_n \leftarrow selectHRT(n, RT)$
4: **for all** x_i such that $1 \leq i \leq n$ **do**
5: $b \leftarrow 100$
6: $k \leftarrow \left\lfloor \frac{N}{b} \right\rfloor$
7: $success \leftarrow false$
8: **while** $k \geq 30$ **and** $success = false$ **do**
9: $S \leftarrow s_{i,1}, ..., s_{i,k} \leftarrow segment(N, b)$
10: $Y \leftarrow y_{i,1}, ..., y_{i,k} \leftarrow maxima(S)$
11: **if** $passChiSquareTest(Y) > 0$ **then**
12: $lwb \leftarrow \frac{b}{2}$
13: $upb \leftarrow b$
14: $b \leftarrow \left\lfloor \frac{lwb + upb}{2} \right\rfloor$
15: **while** $success = false$ **do**
16: $success \leftarrow lwbsearch(b, Y)$
17: **if** $success = true$ **then**
18: $l, s \leftarrow ChiSquareTest(Y)$
19: $est_i \leftarrow wcrtevt(b, l, s)$
20: **end if**
21: **end while**
22: **else**
23: $b \leftarrow 2 \times b$
24: $k \leftarrow \left\lfloor \frac{N}{b} \right\rfloor$
25: **end if**
26: **end while**
27: $upb \leftarrow b$
28: $b \leftarrow \frac{b + \frac{b}{2}}{2}$
29: **while** $success = false$ **do**
30: $success \leftarrow upbsearch(b, Y)$
31: **if** $success = true$ **then**
32: $l, s \leftarrow ChiSquareTest(Y)$
33: $est_i \leftarrow wcrtevt(b, l, s)$
34: **end if**
35: **end while**
36: **end for**
37: $EST \leftarrow est_i, ..., est_n$
38: $rt_{est} \leftarrow min(EST)$
39: **return** rt_{est}

4.2.4 Selecting the Lowest WCRT Estimation

As the last step in *WCRTEVT*, the lowest WCRT estimate is selected as the WCRT estimate on all m data sets. This is also confirmed by the empirical results presented in Section 5.

5 Empirical Results

A validation model inspired by a real industrial control system is constructed with the purpose to investigate how close the response time given by *WCRTEVT* is to the exact WCRT achieved by the simulation optimization-based method, i.e., HCRR in [8]. Moreover, in order to make the model analyzable by using basic RTA, the adhering task execution dependencies are simplified in that the execution time of the tasks is only varied by asynchronous message-passing with the loop bounds manually added to the simulation model. The results of five different methods are showed in Table 1.

Table 1. The results comparison for the MV.

	MC	*MABERA*	*HCRR*	*Basic RTA*	*WCRTEVT*
MV	4332	4332	4332	5982	**4574.556**

Clearly, the WCRT estimation achieved by *WCRTEVT* is 5.6% (i.e., $(4\,574.556 - 4\,332)/4\,332 \times 100\%$) more pessimistic than the exact value derived by HCRR and MC (Monte Carlo simulation), but 23.5% (i.e., $(5\,982 - 4\,574.556)/5\,982 \times 100\%$) less pessimistic when compared to the value obtained by basic RTA. Hence, we believe that *WCRTEVT* has the potential to provide meaningful results, i.e., tighter upper bounds of the WCRT estimation in the analysis of the real-time systems with more complex execution dependencies between tasks.

6 Conclusions and Future Work

This paper has presented ongoing work towards performing response time analysis for system models with intricate execution dependencies between tasks, by using the proposed statistical-based method based on extreme value theory. Specially, we have presented and validated the method by using a model inspired by real industrial control systems, which shows the benefit over basic RTA, in terms of reduced pessimism. Contrary to existing stochastic real-time analysis, the proposed method is not restricted by the assumption that tasks are independent, that the job-level priority is fixed and that the worst-case scenario only happens in the case of the critical instance. As part of future work, the evaluation on models with more complex execution dependencies between tasks will be conducted.

Acknowledgment

This work was supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS.

References

[1] "Website of ABB Group," www.abb.com.

[2] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings, "Fixed priority pre-emptive scheduling: an historical perspective," *Real-Time Systems*, vol. 8, no. 2/3, pp. 129–154, 1995.

[3] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, ser. LNCS, M. Bernardo and F. Corradini, Eds., no. 3185. Springer–Verlag, September 2004, pp. 200–236.

[4] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Times - a tool for modelling and implementation of embedded systems," in *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. London, UK: Springer-Verlag, 2002, pp. 460–464.

[5] "Rapita systems, www.rapitasystems.com, 2008."

[6] D. Decotigny and I. Puaut, "ARTISST: an extensible and modular simulation tool for real-time systems," in *Proc. of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '02)*, 2002, p. 0365.

[7] J. Kraft, Y. Lu, C. Norström, and A. Wall, "A metaheuristic approach for best effort timing analysis targeting complex legacy real-time systems," in *Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 08)*, April 2008, pp. 258–269.

[8] M. Bohlin, Y. Lu, J. Kraft, P. Kreuger, and T. Nolte, "Simulation-based timing analysis of complex real-time systems," in *The 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 09*, August 2009, pp. 321–328.

[9] G. A. Kaczynski, L. L. Bello, and T. Nolte, "Deriving exact stochastic response times of periodic tasks in hybrid priority-driven soft real-time systems," in *Proceedings of 12th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'07)*. IEEE Industrial Electronics Society, September 2007, pp. 101–110.

[10] J. Kraft, "RTSSim - A Simulation Framework for Complex Embedded Systems," Mälardalen University, Technical Report, March 2009.

[11] J. S. J. Beirlant, Y. Goegebeur and J. Teugels, *Statistics of Extremes: Theory and Applications*. Wiley Press, 2004.

[12] S. H. J. Hansen and G. Moreno, "Statistical-based wcet estimation and validation," in *9th Int'l Workshop on Worst-Case Execution Time Analysis*, 2009, pp. 123–133.

A transparent target function and evaluation strategy for complex multi-objective optimization problems

Florian Pölzlbauer
The Virtual Vehicle Competence Center
Graz, Austria
Email: florian.poelzlbauer@v2c2.at

Eugen Brenner, Christian Magele
Graz University of Technology
Graz, Austria
Email: brenner@tugraz.at, christian.magele@tugraz.at

***Abstract*—In the industrial domain, multi-objective optimization problems are wide-spread. Allocating application tasks onto a network of processing nodes (known as the task allocation problem) is one of them. Here, one major challenge is that several other problems (e.g. task scheduling) must be included. In addition, several constraints (e.g. memory) have to be considered. While the aspect of how to solve such optimization problems has been extensively addressed in the last years, the aspect of how to encode multiple objectives into an optimization target has not drawn as much attention. In this work we present a transparent and comprehensible target function and evaluation strategy for complex multi-objective optimization problems. Two novel target formulation methods are implemented and compared to the traditional *weighted sums* approach.**

***Keywords*-multi-objective; optimization; target function; real-time systems; task allocation;**

I. Introduction

In the literature, distributed embedded real-time systems are described by two models. The hardware model consists of processing nodes, which are connected via shared data buses - it can be represented as an undirected graph. The application model comprises tasks, which exchange data via messages passing - it can be represented as a directed, weighted graph. The application model has to be allocated onto the hardware model, in order to form the implementation. This task is known as the *task allocation problem* [1]. During this step several aspects have to be considered: Both the processing nodes as well as the data buses offer limited resources. For each solution of the task allocation problem, a valid scheduling has to be found for both, tasks and the arbitration of the data buses. Data that has to be exchanged via data buses has to be packed into bus frames. This task is known as the *frame packing problem* [2]. In addition, the resulting real-time behavior of the application has to be valid. Meaning: all deadlines have to be met. Therefore, schedulability analysis for tasks [3] and data buses [4] has to be performed.

The task allocation problem is challenging, since it includes several problems, which by themselves form a multi-objective optimization problem. In order to address such multi-objective optimization problems, several aspects are needed:

- Methods and algorithms that can solve the underlying forward-problem. This is mostly done by special purpose simulation environments. The outputs of this step are the characteristics of one particular solution.
- Methods and algorithms that generate instances of the forward-problem. In general, these can be feasible or infeasible solutions. Of course, only feasible solutions are of main interest, but in general one cannot know the nature of the generated solution, unless the forward-problem is solved for this individual. This step may be performed by general-purpose optimization tools, or by problem-specific heuristics.
- Methods and algorithms that are directing the search through the search space. This may be achieved by adapting search direction and step size. This step may be performed by general-purpose optimization tools, or by problem-specific heuristics.
- An optimization target that considers all individual objectives. In literature this is known as target function, quality function, cost function, fitness function and others.
- Methods and algorithms that evaluate the individual solutions due to the chosen optimization target.

While solving such problems has been addressed in the last years, the aspect of formulating the target function has not drawn as much attention. In this work we focus on the aspect of formulating an optimization target out of several objectives, and on how to evaluate this optimization target. After deriving our approach, we apply it to a multi-objective optimization problem from the automotive domain: task allocation, task scheduling & bus arbitration scheduling.

II. Target Function for Multi-Objective Optimization Problems

In industry one tends to use already existing and proven algorithms. That is why the use of general-purpose optimization algorithms is widely accepted. Basically, those algorithms (e.g. genetic algorithm, particle swarm, ...) can only handle a single target function.

In literature several methods have been proposed to address multi-objective optimization problems [5] by the use of algorithms that can only handle a single target function:

- All objectives are combined, in order to form one target function. The combining is mostly done by the use of a weighted sum. The resulting target function takes into account all objectives simultaneously.
- One objective is used as the target function, while all other objectives are used as boundary functions. In this way, only one objective is primarily taken into account.

- The optimization algorithm tries to find non-dominated solutions. These solutions form the Pareto-front. This approach is suitable, if all objectives are contradicting each other (so that the resulting space between the individual objectives is maximal). If the objectives are not contradicting each other – or it is simply not known, how the individual objectives influence each other – this approach may not find good results. Further, this approach does not return one optimal solution, but a list of equal optimal solutions. Thus, this approach is more suitable for design space explorations.

In the following, we will focus on the first approach: combining all objectives into a single target function.

A. Weighted objectives

A common approach is the use of *weighted sums*.

$$f(x) = \sum_{i=1}^{n} (w_i.f_i(x))^2$$

where $f_i(x)$ represent the individual objectives and w_i represent the corresponding weights. Additionally the terms can be squared, in order to prevent negative and positive terms from canceling each other.

This approach is easy to apply. Further it has the advantage that all objectives are considered at the same time. The drawbacks are that the resulting target function $f(x)$ may get non-transparent, since all objectives are mixed together. Further, finding adequate weights is crucial. This task is challenging, since it has to consider both needs:

- user intention, in what extend an individual objective shall be considered
- scaling of the individual objectives in order to be comparable

In order to attain these needs easier, we propose a slightly modified approach by adding an additional factor, leading to:

$$f(x) = \sum_{i=1}^{n} (s_i.w_i.f_i(x))^2$$

with:

s_i	scaling of the individual objectives, in order to meet comparable values
w_i	weighting of the individual objectives, due to the user intention
$f_i(x)$	individual objectives

This modification makes it easier to specify the user intention and scaling of objectives (by making them explicit). Nonetheless, the disadvantages of non-transparent results and the challenge of finding adequate scaling-factors and weights remain. In order to address these challenges, we propose a method to automatically derive adequate scaling-factors (see Algorithm 1). The main idea is as follows:

- Define a range that includes all objective value after scaling
- Determine the maximum possible value of each objective
- Determine the minimum possible value of each objective
- Derive the needed scaling factor

Algorithm 1: Determining adequate scaling factors

Input: $f_{\max}$ desired upper bound of objectives
Input: $f_{\min}$ desired lower bound of objectives
Input: $f_{i,\max}$ max. possible value of objective
Input: $f_{i,\min}$ min. possible value of objective
Output: s_i scaling factor

```
begin
    Δf = f_max − f_min;
    Δf_i = f_{i,max} − f_{i,min};
    s_i = Δf / Δf_i;
end
return s_i
```

By using this simple approach, it is possible to derive adequate scaling factors. The resulting scaling is done linear. The main task is to find the minimum and maximum possible value of the individual objectives. The challenge that still remains, is finding adequate weights. Here, not only the values of the weights are important, but also the ratio between them. Thus, for a large amount of objectives, this task is hard to perform.

Observing some industrial multi-objective optimization problems, we found that objectives from different engineering domains as well as from different engineering detail-levels have to be stated simultaneously. For the *distributed real-time systems* domain typical objectives could be:

- system weight $\rightarrow \min$
- system cost $\rightarrow \min$
- bus utilization $\rightarrow \min$
- node CPU utilization $\rightarrow$ even
- task lateness $\rightarrow \max$
- task response time $\rightarrow \min$

Taking a closer look at those objectives, they can be classified into 3 groups:

1) economic objectives (weight, cost)
2) scalability objectives (utilization)
3) real-time behavior objectives (lateness, response time)

When applying the *weighted sums* approach to these objectives, a trade-off between the individual objectives is established according to the chosen weights. This approach may be adequate, if all objectives are more or less "equal" (in terms of that they all shall be considered simultaneously). In case of different objectives from different engineering domains and detail-levels, this approach may no longer be adequate.

B. Prioritized objectives

When dealing with "unequal" objectives (in terms of them being from different domains and detail-levels) we could observe the following:

- Some objectives are more important than others
- Between unequally important objectives, high-important objectives always overrule low-important ones
- Between equally important objectives, a trade-off is desirable

These demands can hardly be addressed by the use of weights alone. Weights are able to accomplish a trade-off between equally important objectives. When it comes to overruling objectives, another concept can be used: prioritizing.

Instead of assigning weights to each objective, we assign priorities. The priority states the importance of each objective. When evaluating a solution, the objectives are considered due to their priorities: At first the objective with the highest priority is evaluated. If the value of this objective shows an improvement compared to the value of the currently best solution, the new solution wins the evaluation process and becomes the new best solution. If the value of the objective of the new solution shows degradation, the new solution loses the evaluation process. If the value of the objective is equal to the value of the currently best solution, the evaluation cannot decide which one wins. Therefore, the objective which has the next-lower priority is used to determine, which solution wins. For this objective, the same strategy is applied. This evaluation process is repeated until a decision can be made, or all objectives are evaluated.

Algorithm 2: Evaluation of objectives due to their priority

```
Input: x* currently best solution
Input: f[p](x) individual objective with priority p
Input: p priority of objective
Output: x*
1  begin
2      for p = 1 to P do
3          if f[p](x) better than f[p](x*) then
4              x* ← x;
5              exit;
6          else if f[p](x) worse than f[p](x*) then
7              exit;
8          else
9              p++;
10         end
11     end
12 end
13 return x*
```

Applying this evaluation strategy (see Algorithm 2), we can address multi-objective optimization problems where differing objectives (in terms of domain, detail-level, ...) have to be considered. Instead of splitting the problem into several parts and solving those parts sequentially, we can address all objectives within one optimization run. At the same time we can consider the need/demand that objectives can overrule other objectives (instead of using the *trade-off* approach). Another advantage of this evaluation strategy is that scaling factors for each objective are not needed any more, since in every step of the evaluation process only objectives that are of the same kind (and therefore within the same range of values) are compared.

Additionally, the evaluation – and thus also the optimization process – is more transparent (compared to the *weighted sums* approach). Therefore optimization results can be comprehended more easily. This is a very important factor when dealing with real-world industrial problems. The proposed evaluation strategy may therefore be easier accepted within industrial problems.

C. Prioritized weighed objectives

So far we have shown that by applying the concept of prioritizing, we can formulate a target function where objectives can overrule each other. Unfortunately this approach has one major disadvantage. It does not feature trade-offs or *balancing of objectives*. Meaning: accept small worsening of one objective, in order to archive bigger improvements of other objectives (e.g. accepting an increase of 3% bus utilization in order to get a decrease of 10% task response time).

In order to overcome this drawback, we enhance our approach by combining it with the ideas of the *weighted sums* approach. The main idea is as follows:

- We assign to each objective a priority
- If several objectives have the same priority, we additionally assign a weight
- At first we evaluate all objectives which have the highest priority
- If there exist more than one objective at this priority level, we apply the concept of *weighted sums* to them
- If the objectives of this priority level are *better than* the best solution, the new solution wins.
- If the objectives of this priority level are *worse than* the best solution, the new solution loses.
- If the objectives of this priority level are *equal to* the best solution, the objectives of the next-lower priority level are evaluated.
- This procedure is applied until a decision can be made, or all objectives of all priority levels are evaluated.

The priority value is used to group objectives due to their priority/importance. High-priority objectives will always overrule low-priority objectives. Assigning the priority is very intuitive, and thus easy to perform. The weight is used to state the importance of an objective in comparison to equal-priority objectives. Assigning weights is still demanding, but since the amount of equal-priority objectives is smaller than the amount of all objectives, the effort is significantly lower.

This novel approach combines the advantages of both approaches. Unfortunately the use of adequate scaling factors between equal-priority objectives is still needed. In order to reduce the effort for configuring the evaluation strategy, we proposed a way of how the needed scaling factors can be determined automatically (see Algorithm 1). This way the user can focus on assigning adequate priority levels and weights, in order to set up the optimization target.

Based on this novel approach, one can state target functions that are more transparent and thus can better fit industrial multi-objective problems. Applied to our example, the following target function could be addressed:

- Find system configurations that have best economic properties (minimum system weight, minimum system cost)
- Out of those configurations, find those that have best scalability properties (minimum bus utilization, even node CPU utilization)

- Out of those configurations, find those that have best real-time behavior (maximum task lateness, minimum task response time)

In certain cases, this kind of formulation will better fit the intentions of the user. Instead of being forced to express his intention by using weighing factors (that will create a trade-off between individual objectives) one can use both: priorities for stating importance between objectives that can overrule each other, and weights for stating balancing of objectives that do not overrule each other.

III. Preliminary results

In order to evaluate our approach, we have applied it to the task allocation optimization problem. Optimization objectives are: minimum system weight, minimum system cost, minimum bus utilization, even node CPU utilization, maximum task lateness. Constraints are: processing node memory, processing node CPU utilization, bus utilization, deadlines. Task allocations are generated and evaluated following a hill climbing approach.

In order to solve the underlying forward problem, we have made certain simplifications: Tasks are only activated by the arrival of a message. Therefore task scheduling can be performed by rate monotonic (RMS) or deadline monotonic (DMS). As the bus protocol we assume CAN [4]. Therefore, RMS can be applied for the bus arbitration scheduling. For evaluating the real-time behavior we use holistic schedulability analysis [3].

The following problem instance has been evaluated:

- 5 processing nodes, 1 shared data bus
- 14 tasks, 8 messages
- maximum 10.000 iterations

objective	weighted sums	prioritized	prioritized weighted
system weight [kg]	3.0 [w=100]	3.0 [p=1]	3.0 [p=1,w=1]
system cost [$]	168.0 [w=100]	168.0 [p=2]	168.0 [p=1,w=1]
bus utilization [%]	3.93 [w=10]	1.20 [p=3]	3.03 [p=2,w=1]
Δ CPU utilization [%]	4.72 [w=10]	14.15 [p=4]	4.26 [p=2,w=1]
task lateness [ms]	329.61 [w=1]	341.99 [p=5]	338.90 [p=3,w=1]
# improving steps	9	8	7
feasible solutions [%]	4.83	5.27	5.15

TABLE I
Optimization results (median over 50 runs) [w..objective weight, p..objective priority]

As metrics for comparing the different approaches we use the resulting objective values, which are: system weight, system cost, bus utilization, average Δ CPU utilization, average task lateness.

The most striking fact is that all approaches get equal results for *system weight* and *system cost*. The reason for this is that (in our model) weight and cost can only be improved by detaching processing nodes or data buses. This is hard to achieve, due to high CPU utilizations and tight task deadlines. The *prioritized* approach gets by far best *bus utilization* and worst Δ *CPU utilization*. Results for *task lateness* are more or less equal for all approaches. Both the *weighted sums* approach and the *prioritized weighted sums* approach get comparable results for *bus utilization* and Δ *CPU utilization*, however, the *prioritized weighted sums* approach is about 30% better for *bus utilization* and about 10% better for Δ *CPU utilization* compared to the *weighted sums* approach. All approaches find about 5% feasible solutions. Additionally it can be seen that (out of the 483 to 527 feasible solutions) only 7 to 9 solutions result in an improvement of the optimization target.

It can be seen that both, the *weighted sums* approach and the *prioritized weighted sums* approach can be used for stating equivalent target functions for multi-objective optimization problems. The main challenge when using the *weighted sums* approach is to find adequate weights. The *prioritized weighted sums* approach provides a more transparent and intuitive way to formulate the optimization target. This is especially useful for high number of objectives.

IV. Conclusion & Outlook

In this work we have presented a novel approach to formulate a target function for multi-objective optimization problems. The approach allows stating both overruling objectives (by using the concept of prioritizing) as well as balancing of objectives (in order to find trade-offs between equal-prioritized objectives). The approach is intuitive and flexible. Thus it provides a more transparent and comprehensible target function for multi-objective optimization problems.

In future research activities we will continuously enhance the task allocation problem. The main focus will be put on scheduling of mixed (time-triggered & event-triggered) tasks and scheduling of automotive buses (LIN and FlexRay). In addition, we will evaluate different stochastic optimization algorithms (including particle swarm).

Acknowledgment

The authors wish to thank the COMET K2 Forschungsförderungs-Programm of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Economics and Labour (BMWA), Österreichische Forschungsförderungsgesellschaft mbH (FFG), Das Land Steiermark, and Steirische Wirtschaftsförderung (SFG) for their financial support.

References

[1] P. Pop, P. Eles, Z. Peng, and T. Pop, "Analysis and Optimization of Distributed Real-Time Embedded Systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 593–625, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1142980.1142984

[2] R. Saket and N. Navet, "Frame packing algorithms for automotive applications," *Journal of Embedded Computing*, vol. 2, no. 1, pp. 93–102, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1370995

[3] K. W. Tindell, A. Burns, and A. J. Wellings, "An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks," *Real-Time Systems*, vol. 6, no. 2, pp. 133–151, 1994.

[4] L. Pinho, F. Vasques, and E. Tovar, "Integrating Inaccessibility in Response Time Analysis of CAN Networks," in *Proceedings of IEEE International Workshop on Factory Communication Systems*, 2000, pp. 77–84.

[5] T. Weise, *Global Optimization Algorithms - Theory and Application*, 2nd ed. Self-Published, Mar. 3, 2009. [Online]. Available: http://www.it-weise.de/

Prediction-based Interrupt Scheduling

Yuting Zhang
Computer Science Department
Merrimack College
zhangy@merrimack.edu

Abstract

Interrupt scheduling is a critical factor in achieving predictable real-time service. Since I/O interrupts are usually generated on behalf of specific processes, they should be scheduled in accordance with the priorities of processes that request them. However, the challenge is how to identify the requesting process for a given I/O interrupt before handling it. This paper explores the effect of I/O interrupts on real-time process scheduling, and proposes three prediction schemes that leverage the historical data to identify the urgency and importance of executing a deferrable interrupt handler. The preliminary performance data has shown the effectiveness of this approach on a modified Linux system.

1 Introduction

Interrupt scheduling is a critical factor in achieving predictable real-time service. In most operating systems, interrupt handling is largely independent of process scheduling. However, the handling of interrupts inevitably affects process execution. In particular, I/O interrupts are mostly initiated as a result of I/O requests from processes. To avoid significant impact to process scheduling and execution, we argue that I/O interrupts should be scheduled and accounted, where possible, on behalf of corresponding processes [10].

However, it is not usually known and not trivial to identify which process corresponds a given interrupt before handling it. This is particularly true for network reception interrupt, where the requesting process for an incoming packet cannot be identified until the end of the interrupt service. While early demultiplexing has been applied to ATM networks [1], where hardware support provides a method of identifying the processing end-point for a given interrupt [1], it is difficult to perform the same approach with Ethernet controllers. Similarly, other I/O device controllers, such as SCSI or SATA disk controllers, do not provide an obvious means of associating interrupts with specific processes. A network interrupt from an Ethernet device could possibly be dealt with by a top half [2] that strips headers from packet data in main memory, until socket identification is obtained. However, this not only requires modification to device drivers, which is itself an unenviable task, but it may unduly extend the execution time of top halves. It should be noted that top halves often run with interrupts of the same or lower priority being disabled, so subsequent interrupts may be lost if top halves do not complete quickly.

Instead of changing the handling of top halves, our work has so far focused on the scheduling and accounting of the bottom halves [10]. We have proposed a simple priority-based scheme in which a bottom half's priority is set to the highest priority of all processes waiting on the corresponding I/O device for a given interrupt. This simple scheme may work in some scenarios. However, it mispredicts the priority of the majority interrupts when the highest-priority blocked process is actually associated with infrequent interrupts from a particular device.

Given the above challenges, this paper proposes three prediction schemes to schedule I/O interrupts that leverage historical information collected over a finite sliding window. The proposed prediction schemes avoid changes to device drivers or top half handlers, but instead attempt to predict the urgency and importance of servicing a pending bottom half for a given hardware interrupt. Some preliminary experiment results demonstrate the effectiveness and simplicity of the proposed schemes on the modified Linux network interrupt handler.

The remainder of the paper is organized as follows: Three prediction algorithms are described in Section 2. Section 3 shows the experimental evaluation in Linux. Related work is described in Section 4, followed by conclusion and future work in Section 5.

[1] With ATM networks a communication connection can be identified by a virtual circuit, or virtual path, identifier.

[2] Typical general purpose systems such as Linux split interrupt service routines into "top" and "bottom" halves. The top half performs only the basic service requirements at the time of the interrupt.

2 Prediction Algorithms

To identify the corresponding process for a given interrupt, the first step is to find all possible candidates. Though a candidate can be any process waiting on the same device that generates the given interrupt, as described in [10], those who have recently received interrupts would more likely be the requesting process in general. The historical data can be used to help make better prediction.

Whenever an interrupt is handled, the information of its associated process and the time stamp can be obtained. Each device maintains such information for the last N interrupts processed. In addition, it keeps track of the time at which the latest interrupt is handled, denoted as t^l. Moreover, for each process P_i that is waiting on the device, the collected data includes the number of the interrupts handled for P_i out of the last N interrupts, denoted as n_i, and the time at which the latest interrupt is handled for P_i, denoted as t_i^l. With both time and frequency information, the candidates can be defined as the recently active processes that have received any of the last N interrupts within the past Δ time units, where N and Δ are two configurable parameters. The accuracy of the prediction can be enhanced by this refined candidate set. To reduce the overhead, a circular array of size N can be used to implement this sliding window buffer for each device.

Figure 1 shows the algorithm of updating interrupt history for a particular device and the related processes, where t is the current time, H is the historical data array, P_r is the requesting process of the current interrupt, end is the ending index of the circular array H, $count$ is the real number of the records in the array. S is the requesting process candidate set.

Algorithm *Updating Interrupt History*
Input: P_r
1. **if** ($H[end]$ records a valid process P_i)
2. $count--$; n_i--;
3. **if** ($n_i == 0$) delete P_i from S;
4. add P_r into $H[end]$;
5. $count++$; $t^l = t$; $end = (end+1)\%N$;
6. **if** (P_r is a valid process)
7. **if** ($n_r == 0$) add P_r into S ;
8. n_r++; $t_r^l = t$;

Figure 1. Updating Interrupt History

The next step is to infer the requesting process from all candidates based on the above historical information. Three different prediction schemes are described below.

Highest Priority Based Prediction Scheme: Essentially, this is our previously published scheme [10], which associates the given interrupt with the highest priority process waiting on the same device that generates the interrupt. However, the difference is that the candidates here are not all processes that have requested interrupts from a specific device, but only those processes that have received one or more of the most recent N interrupts. In addition, the time constraint Δ is used to filter out the candidates that have remained inactive for a long time.

Highest Frequency Based Prediction Scheme: It may be the case that the highest priority process is not the one requesting the current interrupt, nor any interrupts in the backlogged queue. For example, the interrupts in the current backlogged queue are all requested by a lower-priority process. Consequently, another prediction scheme is proposed that determines the requesting process based on the *frequency* of the interrupts handled for each candidate process. The interrupt frequency of a process P_i is the number of interrupts handled for it among the last N interrupts, denoted as n_i above. The intuition behind this scheme is that the interrupts in the backlogged queue may be most probably requested by the process that has the highest number of interrupts handled recently.

Frequency Based Probabilistic Prediction Scheme: Instead of using a deterministic approach, this scheme assigns the probability of being the requesting process for each candidate according to its interrupt frequency. Specifically, the probability of each process is calculated as the ratio between the number of interrupts handled for it among the last N interrupts and N. The simple intuition behind this is that the process which has more interrupts handled in the past has a higher probability of requesting future interrupts. Figure 2 shows the algorithm for the frequency-based probabilistic prediction scheme.

Algorithm *Frequency Based Probabilistic Prediction*
Input: S
Output: P_p
1. Generate a uniformly random number r in $(0, N]$;
2. $P_p = NULL$;
3. $curTot = 0$;
4. **for** each P_i in S
5. $curTot += n_i$;
6. **if** (($curTot \geq r$) && ($t^l - t_i^l \leq \Delta$))
7. $P_p = P_i$; return P_p;
8. return P_p;

Figure 2. Frequency Based Probabilistic Prediction

For a given interrupt, the requesting process can be identified using any of the above prediction schemes, and the priority of that interrupt handler is set to the priority of the predicted process. It is then compared to the priority of the current running process. If its priority is higher than that of the current process, or the inferred requesting process is the currently executing process, the bottom half is exe-

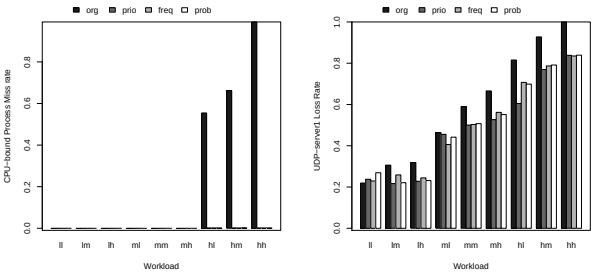
cuted immediately. Otherwise, it is deferred until the next scheduling point.

Ideally, the scheduling decisions should be made for each interrupt that requires handling. However, this typically causes high overhead. Usually in general operating systems such as Linux, all backlogged interrupts are handled together in one bottom half invocation. Therefore, it is more practical to make the scheduling decision every time a bottom half is invoked for all backlogged interrupts.

3 Experimentental Evaluation

The proposed approach with all three prediction schemes is implemented upon Linux network handler for packet reception interrupt. Different combinations of periodic UDP-server and CPU-bound processes are used in the experiments to evaluate the performance of the system with each of the three prediction schemes implemented in interrupt handler (priority-based scheme is denoted as "prio", frequency-based scheme as "freq" and probabilistic scheme as "prob" in the figures below) against the original Linux system (denoted as "org"). For each UDP-server process, a UDP-client process is running on a separate machine to periodically send a fixed number of packets to it. Both UDP and CPU-bound processes are set in SCHED_FIFO class. The higher priorities are assigned to the processes with the shorter periods. This essentially simulates Rate Monotonic Scheduler (RMS) in Linux. All experimental machines have 1.80GHz Pentium IV processors with 1G RAM connected by a PRO/100Mbps Ethernet cards.

First, a single UDP-server process in low priority (UDP-server1) and a CPU-bound process in high priority (CPU-bound) are running on the experimental machine. In Figure 3, the horizontal axis represents the workload consisting of two letters. The first letter is the workload of CPU-bound, and the second one is of UDP-server1, where l starts for low, m for medium, and h for high. As shown in Figure 3 (a), CPU-bound has no deadline misses on the modified Linux with any prediction scheme under all different workloads. By contrast, deadline misses start occurring on the original Linux when the workload of CPU-bound is high, and the miss rate is up to 1 when the workload of both CPU-bound and UDP-server1 are high. While UDP-server1 has lots of losses on all systems due to its low priority, there are much more losses on the original Linux than the modified Linux with the prediction schemes in most cases as shown in Figure 3 (b). All three prediction schemes have similar performance because there is only one UDP process with active network interrupts all the time. Several other system processes, such as ntpd (low priority), ypbind (low priority) and watchdog (high priority) may receive the packets from the same network device occasionally, which causes the slight difference between different prediction schemes.



(a) (b)

Figure 3. Single UDP-server Process

On the original Linux, the service for CPU-bound is frequently interrupted by processing the incoming packets. Though most packets are handled by the bottom half, many of them will then get dropped afterwards because UDP-server1 in low priority cannot be scheduled to pick them up in time. The wasted service time for these dropped packets in bottom half degrades the service for both CPU-bound and UDP-server1. However, on the modified Linux with any of the prediction schemes, the interrupts are associated with UDP-server1, and thus the low priority of UDP-server1. Consequently,the interrupt handler gets deferred frequently and this delay enables the guaranteed service for CPU-bound. Moreover, such delay may also cause some packets to be dropped before bottom half invocation, or packets to be handled in batch, which reduces the wasted time and overhead in bottom half. In return, UDP-server1 receives better service.

Then, two UDP-server processes, UDP-server1 in lower priority and UDP-server2 in high priority, are running on the experimental machine with a single CPU-bound process in medium priority. In Figure 4, the first letter of the workload representation by the horizontal axis is of UDP-server2, and the second one is of UDP-server1. The workload of CPU-bound keeps as medium. On the system "prio", the interrupt handler is mostly associated with the high priority of UDP-server2 and thus executed immediately to receive the packets, which results in better service for UDP-server2, but worse service for CPU-bound and UDP-server1 as shown in Figure 4. This is closer to the performance on the original Linux, where the interrupt service has the highest priority in general. On the system "freq", when the workload of the UDP-server2 is lower then UDP-server1, the interrupt handler is mostly associated with the low priority of UDP-server1 and thus the execution gets deferred. Consequently, this results in better service for CPU-bound and UDP-server1, but may cause more packets dropped unnecessarily for UDP-server2 as shown in Figure 4(d) under workload "lm", "lh", "mh" and "hh". On the system "prob", the priority of interrupt handler is probabilistically changing based on the interrupt frequency

of UDP-server1 and UDP-server2. As shown in Figure 4, it provides much better service for CPU-bound and UDP-server1 than the system "prio" and "org", but this performance gain is at the certain cost of service for UDP-server2 as on the system "freq". However, it provides much better service for UDP-server2 than the system "freq". Moreover, Figure 4(a)(b) shows that overall the system "prob" has the lowest CPU miss rate and the lowest total UDP-server loss rate out of all four systems. By using frequency based probabilistic prediction scheme, the interrupt scheduling can be more intelligently coordinated with process execution and therefore result in better performance.

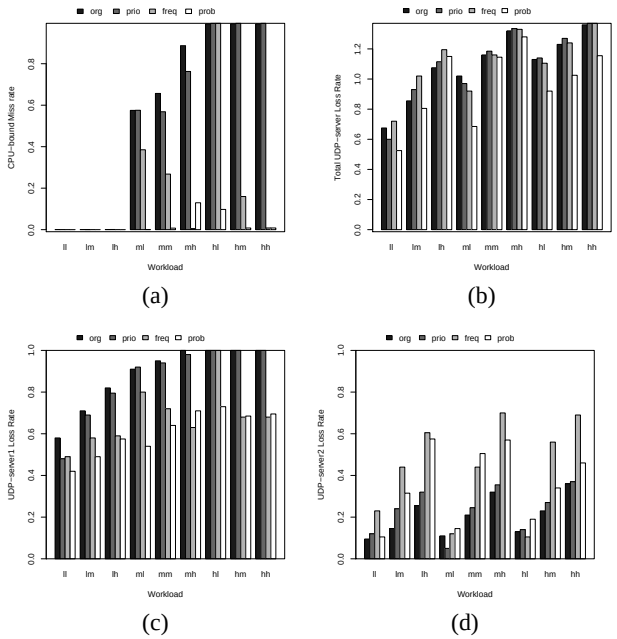


Figure 4. Multiple UDP-server Processes

4 Related Work

Many researchers have looked at the effect of interrupt handling on the process execution. Mogual and Ramakrshnan studied the overload behaviors in the interrupt-driven network subsystems and proposed a set of techniques to eliminate receive livelock [4]. Research efforts such as RTLinux [9] and RTAI [8], provide real-time support on off-the-shelf (COTS) system through separated real-time interrupt handlers. Other approaches provide predictability by imposing a CPU time budget on interrupt service [2, 3, 7]. However, none of the above research works distinguish the interrupts requested by different processes and schedule the interrupt handlers based on the corresponding processes.

Early demultiplexing [1, 5, 6] has been used to identify the relationship between network interrupts and the associated processes. While this is important to provide high-performance network, this in general requires special hardware support or network driver modification. In contrast, this work attempts to provide fairer and more predictable service by *predicting* the dependencies between interrupts and processes based on the historical information and schedule them accordingly without any addition hardware support or driver modification.

5 Conclusions and Future Work

This paper explores the effect of I/O interrupts on the process execution, and proposes three prediction schemes to identify the corresponding process associated with an I/O interrupt. The preliminary experiment results have shown the possible performance gain of the proposed approach, especially with the probabilistic prediction scheme. More evaluation is to be done. Currently, this work is being implemented on Linux with Real Time kernel patch. The evaluation is under the way. Other future work includes evaluating this approach in the SMP system and developing an analytic model for the probabilistic prediction scheme.

References

[1] P. Druschel and G. Banga. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 1996.

[2] T. Facchinetti, G. Buttazzo, M. Marinoni, and G. Guidi. Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In *17th Euromicro Conference on Real-Time Systems*, 2005.

[3] M. Lewandowski, M. J. Stanovich, T. P. Baker, K. Gopalan, and A.-I. A. Wang. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 57–68, Washington, DC, USA, 2007. IEEE Computer Society.

[4] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, 15(3), 1997.

[5] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 1987.

[6] G. Parmer and R. West. Predictable interrupt management and scheduling in the composite component-based system. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 232–243, Washington, DC, USA, 2008. IEEE Computer Society.

[7] J. Regehr and U. Duongsaa. Eliminating interrupt overload in embedded systems. Unpublished.

[8] Real-Time Application Interface: http://www.rtai.org.

[9] Real-Time Linux: http://www.rtlinux.org.

[10] Y. Zhang and R. West. Process-aware interrupt scheduling and accountability. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, Rio de Janeiro, Brazil, December 2006.

On Interrupt Scheduling based on Process Priority for Predictable Real-Time Behavior

Minsub Lee, Juyoung Lee, Andrii Shyshkalov, Jaevaek Seo, Intaek Hong, Insik Shin
Dept. of Computer Science
KAIST, South Korea
insik.shin@cs.kaist.ac.kr

Abstract

Traditionally, kernel services are of a higher priority than user processes. The kernel can preempt the currently executed process in order to perform interrupt handling for the behalf of another process, even though the latter process is of a lower priority than the former. This can be viewed as priority inversion. We propose a new interrupt handling approach that couples interrupt scheduling with the priority of a process associated with the interrupt to handle. We present techniques to derive exact process priorities in handling interrupts for incoming network packets. The proposed approach has been implemented in Linux 2.6, and experiment results show that it reduces interference of lower priority processes to higher-priority process through interrupt handling.

1 Introduction

As the number of hardware devices grows higher, general purpose operating systems are more often used for real-time applications. Such operating systems were not originally designed to satisfy real-time application requirements. Therefore, a number of studies have been conducted to add predictable and efficient task management to commodity operating systems [8],[3].

One of the main design goals of commodity operating systems is the system's responsiveness. Particularly, it implies that interrupt handling must be processed by an operating system immediately. However, interrupts processing, which can take some time, may not be useful for the currently executed task. Additionally interrupt processing time is often deducted from interrupted thread. Because interrupted thread does not necessarily get affected by interrupt processing, this can dramatically distort the real-time performance of the operating system. Therefore, interrupts handling is an important issue to address in order to provide predictability on the execution of real-time tasks.

In Linux, interrupts are processed by kernel threads, which have higher priority than any user thread. From thread scheduler point of view, scheduling is done according to threads' priorities. However, because interrupts are processed to serve different threads with different priorities, the priority inversion may occur. As an example, consider a running user process with priority 17, a sleeping one with priority 21 and an incoming network packet for the second process. This packet will be processed immediately by kernel process, which in turn will delay a thread of higher priority. The more incoming packets, the less predictable is execution of the first thread.

In this paper we addressed the priority inversion problem, which is caused by interrupt processing of network stack. We have designed a technique to process the network interface card interrupts in order of priorities of the threads that require interrupts processing. We have also implemented our technique for Linux kernel version 2.6 and conducted performance evaluations. Our results show that our interrupt handling approach is suitable to real-time environment.

The rest of paper is organized as follows. Section 2 summarizes related work. Section 3 gives an overview of the interrupt handling in Linux. Our approach is described in section 4 and quantitative performance evaluation of our implementation is provided in section 5. Finally, section 6 contains conclusions.

2 Related Work

Several other research projects have investigated interrupt processing distortions on real-time performance of Linux systems. An improved accounting of consumed CPU time during interrupts has been proposed in [3] and [8]. A probabilistic approach [8] has been developed to determine possibly affected processes during top half execution and schedule bottom half with regard to priorities. In this paper however, we describe an approach to find exact processes, and hence priorities, for each interruption caused by incoming network packet. Therefore, our approach can make priority based interrupt scheduling decisions more accurately.

Some researchers focused on other systems than Linux to investigate on similar issues. Scheduling of interrupts

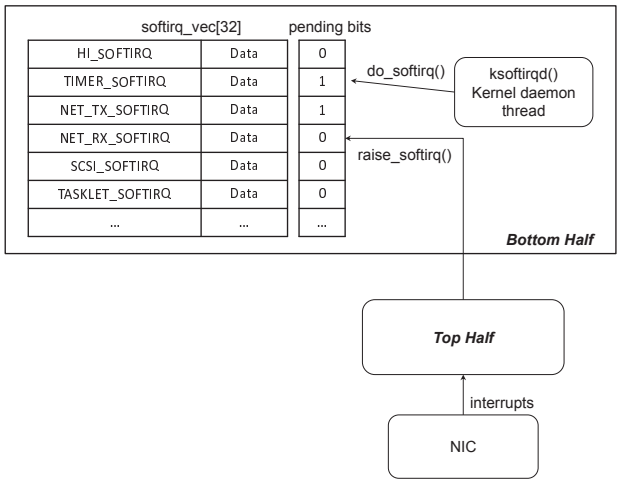


Figure 1. Top and bottom halves of interrupt handling in Linux

and predictable interrupt management has been developed for complex systems [5]. Improved performance of network stack in UNIX operating systems has been proposed by scheduling incoming network traffic with priorities [2]. While these studies focused on providing fairness and increased throughput under high load, our technique focuses on real-time behavior.

Many protocols have been introduced to address the priority inversion problem when tasks are accessing critical sections in a mutually exclusive manner. Such protocols include the Priority Inheritance Protocol (PIP) [7], the Priority Ceiling Protocol (PCP) [6], and Stack Resource Policy (SRP) [1]. While these protocols concern the priority inversion problem within the context of process scheduling, our work concerns the problem taking process scheduling and interrupt handling into consideration together.

3 Interrupt Handling

Interrupts can be caused by hardware as well as by software. In Linux, interrupt handling is done by the kernel, which is invoked every time an interrupt occurs. Interrupts can occur at any time during execution, their number is difficult to predict.

To achieve better performance and responsiveness, interrupt handling in Linux kernel 2.6 is divided into two phases. The first one, called *top half*, starts when an interrupt signal invokes the interrupt service routine (ISR). Then, ISR disables interrupts of the same type and calls the corresponding interrupt handler. Because interrupt handlers execute asynchronously, the processing at this phase should be as quick as possible.

For instance, upon receiving incoming packets off the network, network interface cards (NIC) issue interrupts immediately to alert kernel of their availability. Then, the ISR quickly responds to the interrupts by executing the network card's registered handlers. Most of all, they copy the new packets into the main memory. For remaining processing, network card's registered handler must raise software interrupt (`softirq`) [4], which means setting pending bit to 1 in a softirq vector array (`softirq_vec`) (Figure 1).

All other interrupt processing is deferred to later, so-called the *bottom half* phase. Bottom half usually requires longer processing time than top half. Under heavy load such as high network traffic, the frequency of interrupts is high, and bottom half processing can consume much of the CPU time.

In Linux, bottom half phase is executed periodically by a set of per-processor kernel threads (`ksoftirqd`), which have priority of 15. These kernel threads scan softirq vector array for set pending bits and execute corresponding handlers for further interrupt processing. Additionally, interrupts are processed regardless of the priorities of the processes, which interrupts serve.

Consider a currently running user process with the highest priority, which does not have any network communication. Any incoming network network packet will interrupt the current thread at least once during top half phase processing. Later, since the kernel thread has even higher priority, the bottom half processing may interrupt current thread for even longer time than top half, if the packet is large. Therefore, since the packet is processed for another process with lower priority than current, such scenario is a priority inversion one.

Since the top half cannot be delayed, in this paper we are particularly interested in determining when is the proper time for executing the bottom half phase and which interrupts to process first.

4 Our Approach

This section describes the design and implementation of our new interrupt handling approach in Linux. A key idea of our approach is that some incoming packets are associated with its intended receiver processes (and their priorities) during interrupt handling. Those packets are given process priorities in the top half, and the bottom half is equipped with priority-based scheduling capable of delaying interrupt processing even further, in order to avoid the priority inversion problem.

UDP packets carry process-related information, which is a port number. In the top half, an UDP packet is fetched from the NIC's buffer to the main memory. We then extract a port number out of the packet. This can be easily done (with a single memory lookup), as the port number is stored in a fixed location in the packet. In order to assign priorities to packets, it needs to figure out which sockets are

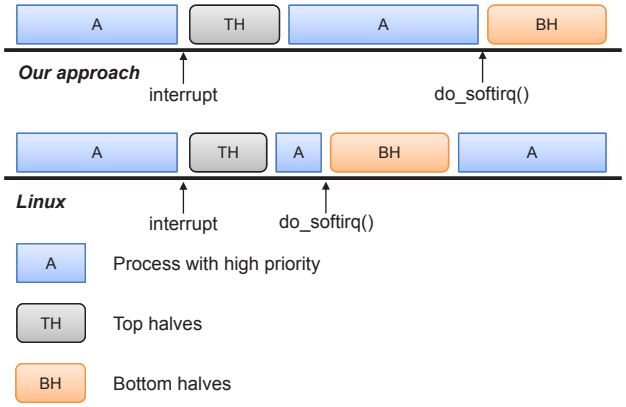


Figure 2. Linux vs. our approach

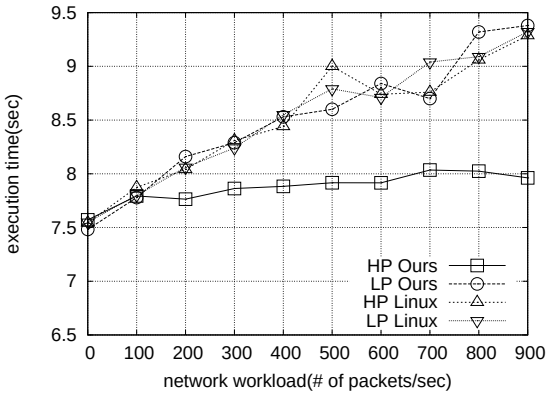


Figure 3. Comparison of execution times

coupled with which processes, which is costly. Hence, we maintain a port number indexed process priority table, and it helps to achieve a faster conversion of a port number to a priority. Each entry of the table is created when a process binds a socket and becomes invalid when the process closes the socket. Once a port number is available, we can simply consult this table to map the port number to the priority of a corresponding process. Then, ISR places the packet into the softirq vector array (`softirq_vec`) according to its corresponding priority. Packets are stored in the queue in a decreasing order of priorities. The further processing of a packet is then deferred to the bottom half.

In the bottom half, a kernel thread (`ksoftirqd`) periodically checks out the softirq vector array for set pending bits. When the pending bit for network packet reception is set, our modified `ksoftirqd` goes through the packet data queue of `softirq_vec` to handle packets one by one, as long as their corresponding priorities are no lower than the priority of the currently executed process. Note that the packet data queue is sorted according to packet's corresponding priorities. When our modified `ksoftirqd` meets a packet with a priority lower than that of the currently executed process, it stops taking care of incoming packets.

Figure 2 shows our new interrupt service approach, in comparison to the original Linux. By using our interrupt service routine, we can reduce interference of interrupt handling to processes by as much as $T_b * N_l - T_d * N_h$, where T_b is the bottom half executing time, T_d is the extra time cost in top half for early demux, and N_h and N_l are the number of packets for higher and lower priority processes, respectively. According to its design, top half extra work is much smaller than that of bottom half, so interference to processes can decrease when N_l is smaller then N_h. It means our interrupt service model can provide less interference to higher priority process.

5 Experimental Evaluation

This section presents experimental results in order to show that our approach is suitable in the real-time environment.

5.1 Experiment Setting

To implement our approach, we make minimum modifications to the Linux kernel and network device driver. We patched Linux kernel 2.6.23. All experiments are performed on the QEMU emulator with a 2.8GHz x86 single core processor and virtual NIC, which is interconnected with a host machine. The host machine has 2.8GHz AMD Phenom CPU, 4GB main memory.

There are two processes executing concurrently. They are a UDP server and a dummy job process. The UDP server handles burst of the packets, and the dummy job process executes Algorithm 1. We set the priority of the UDP server to 20, which follows default Linux settings. We measure execution time of the dummy job processes with priority 17(LP) and priority 21(HP), which are higher and lower than UDP server, respectively. We use a UDP packet as a network workload. The UDP packets are sent from a host machine through the virtual network device. We perform experiments in the original Linux and in our patched Linux, respectively.

Algorithm 1 Dummy Job Algorithm

```
start_time ← current_clock();
i ← ∞;
while(i--) { i←(i+i)/2+(i-i)/2; }
end_time ← current_clock();
```

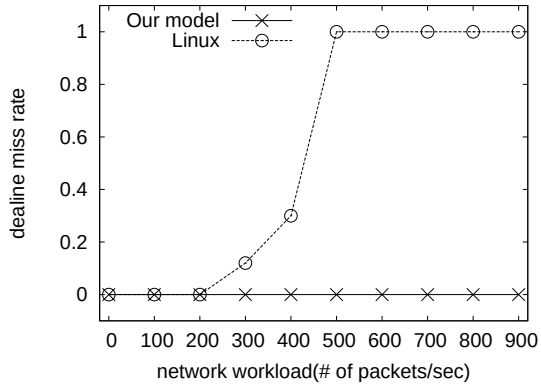


Figure 4. Comparison of deadline miss ratios

5.2 Results

First, we compare execution times of processes with different priority under different network workload. Figure 3 shows measured execution times with the increasing size of burst of packets. In the original Linux, the higher priority process (HP) and the lower priority process (LP) show similar behavior. The reason why two processes show similar behavior is that the network interrupt handling routine preempts both processes without regard to their priorities. Under our interrupt handling approach, the higher priority dummy job process (HP) shows stable execution times. This means that HP is less affected from the interrupts which are associated with the lower priority process (LP). In the top half, the early demux procedure assigns the priority of a receiver process to a bottom half interrupt handling routine. By using this information, the bottom half scheduler is able to "delay" interrupt handling if it is intended for the process with a lower priority than the current process. Such delaying bottom half interrupt handling can reduce the number of times that HP should yield to the interrupt handler. The execution time gap between under original Linux and under our patched Linux process represents how much of bottom halves has been delayed.

In our interrupt service approach, the execution time of HP is only increased about 6% when the client sends 900 packets per second. While the higher priority process shows stable behavior over different network workload, the lower priority process shows similar behavior to other processes under the original Linux setting.

Second, we compare the deadline miss ratio of processes of higher priorities than the UDP server in both Linux and our patched Linux. To measure the deadline miss ratio, we execute the dummy job periodically every 8 seconds and its deadline is 8 seconds. Figure 4 shows the result of this experiment. Under our interrupt service approach, it misses no deadlines. This shows that our interrupt service model is suitable to real-time environment, as it can reduce the interference from lower priority processes through interrupt handling.

6 Conclusion

This paper presents the design and implementation of a new Linux interrupt handling approach for incoming packets. It couples packets with the priorities of their receiver processes, and their interrupt handling is performed according to priorities. This approach is able to prevent the priority inversion problem, in particular, between the currently executed process and the receiver process of a packet under interrupt handling.

We demonstrate the effectiveness of this approach by implementing it over Linux. Experiments show that it effectively provides the predictable execution of processes of higher priorities. In this paper, only UDP packets are covered. Our future work includes accommodating more sophisticated protocols, such as TCP. While TCP employs flow control, delaying interrupt handling can defer TCP acknowledge and may put the TCP communication unstable. We plan to incorporate TCP packets addressing such concerns.

Acknowledgement

We thank anonymous reviewers for their constructive comments. This research was supported in part by IT R&D program of MKE/KEIT of Korea [2009-KI002090, Development of Technology Base for Trustworthy Computing], and KAIST ICC, KIDCS, KMCC, OLEV, and URP grants.

References

[1] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[2] P. Druschel and G. Banga. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation*, 1996.

[3] K. J. Jung, S. G. Jung, and C. Park. Stabilizing execution time of user processes by bottom half scheduling in linux. In *Proc. of Euromicro Conference on Real-Time Systems*, 2004.

[4] R. Love. *Linux Kernel Development*. Novell Press, 2005.

[5] G. Parmer and R. West. Predictable interrupt management and scheduling in the composite component-based system. In *RTSS*, 2008.

[6] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS*, 1988.

[7] L. Sha, J. P. Lehoczky, and R. Rajkumar. Task scheduling in distributed real-time systems. In *Proceedings of the International Conference on Industrial Electronics, Control, and Instrumentation*.

[8] Y. Zhang and R. West. Process-aware interrupt scheduling and accountability. In *RTSS*, 2006.

Time-Based Intrusion Dectection in Cyber-Physical Systems

Christopher Zimmer, Balasubramanya Bhat, Frank Mueller
North Carolina State University
{cjzimme2,bbhat}@ncsu.edu,mueller@cs.ncsu.edu

Sibin Mohan
University of Illinois at Urbana Champaign
sibin@cs.uiuc.edu

***Abstract*—Embedded systems, particularly those with temporal constraints known as real-time systems, are increasingly deployed in every day life. Such systems that interact with the physical world are also referred to as cyber-physical systems (CPS). These systems are common in critical infrastructure from transportation to health care. They impact our life and the environment we live in. While security in CPS-based real-time embedded systems has been an afterthought, security aspects are becoming critical as these systems are increasingly networked and exhibit distributed inter-dependencies. The advancement in their functionality has resulted in more conspicuous interfaces, which can be exploited to attack such systems. Hence, security functionality is becoming a necessary component of embedded real-time design, particularly in the CPS realm.**

In this paper, we present a method for time-based intrusion detection. More specifically, we detect the execution of unauthorized instructions in CPS environments with real-time constraints. The functionality in this work is provided through the utilization of values attained from performing worst-case timing analysis. Timing analysis values are readily available as they are determined prior to the schedulability analysis of real-time systems. Using the same tools that provide a macro view of timing within a CPS application, we demonstrate how to provide more focused timing values for specific execution scopes of an application. Utilizing such focused values, the application is enhanced to engage in internal self checks. Internal timing checks are verified against focused timing values to enable the detection of code injection attacks. To the best of our knowledge, such detection of system compromises through micro-timing information is a novel contribution to CPS environments with real-time constraints.

I. Introduction

Embedded systems have permeated into every aspect of day-to-day life. Examples range from non-critical systems, such as televisions or toasters, over moderately critical systems, such as stop lights or other enhancing infrastructure, to highly critical ones, such as anti-lock breaks, hydro-electric dam controls and flight control systems. The latter two categories are examples of cyber-physical systems (CPS) where system control affects human lives or interacts with the environment in general. Most such cyber-physical control systems are embedded systems with real-time constraints. As these systems are increasingly used in our daily life, insuring that these devices are secure from intrusion and tampering by adversaries is a design challenge of utmost importance.

While the development of real-time systems for the CPS domain is very stringent, there might be vulnerabilities exposed by libraries or methods that may enable an attacker of the system to execute arbitrary instructions on the target machine, *e.g.*, by injecting malicious code. As more embedded applications, particularly CPS applications, utilize networks these attacks are prone to become prevalent against real-time systems as well.

The design constraints of embedded real-time systems lend themselves well to the development of security methodologies while such techniques would not be directly applicable to general-purpose applications. The primary constraint of interest is the detailed knowledge obtained from timing analysis on CPS applications within real-time systems. Here, analysis is performed to determine timing information about the application, such as worst case execution time (WCET) and best case execution time (BCET). These two timing metrics represent a subset of knowledge common to real-time applications, which lend themselves well to security analysis: As WCET and BCET safely bound the upper and lower execution time of specific code sections, execution times above or below the respective bounds are strong indications for a system compromise.

In this paper, we present a methodology that utilizes instrumentation and analysis from within real-time applications in the attempt to detect the execution of unauthorized code. Using actual timing metrics and comparing them with worst-case measurements allows the programs to detect security breaches due to intrusion within the system as well as situations where an application is going to exceed its timing requirements prior to the actual deadline miss, which provides ample time to transition to a fail-safe state.

II. Timing Analysis

Timing analysis is a strict requirement for hard real-time systems where a missed deadline may render the entire system incorrect. Timing analysis is used to insure that an application's best and worst case times can be bounded. The analysis allows designers to verify if system tasks can meet their deadline.

The purpose of timing analysis in real-time systems is generally to determine the schedulability of a task set, *i.e.*, to ensure that each task meets its deadline. In this context, the overall WCET bound of a task becomes the key metric. Our work heavily relies on WCET bounds, but for security reasons and not for the determination of schedulability.

To conduct our study, we use our WCET tool chain [3], [6], [5] that enables us to accurately gauge the WCET values

of several applications from both the macro view of the application as well as micro ranges of instructions in the code. These analysis tools provide timing data at multiple levels and enable the evaluation of such data for more focused ranges of code. Figure 1 depicts a graphical representation of the tools utilized to perform timing analysis in our experiments. A compiler provides an assembly file of the application in annotated PISA assembly format. This intermediate code along with loop bounds is then fed into a control-flow analysis tool. Subsequently, control-flow analysis and static-cache analysis are performed. The respective outputs are then consumed by a timing analyzer. The framework utilizes the annotated assembly and loop bounds to derive safe WCET and BCET bounds.

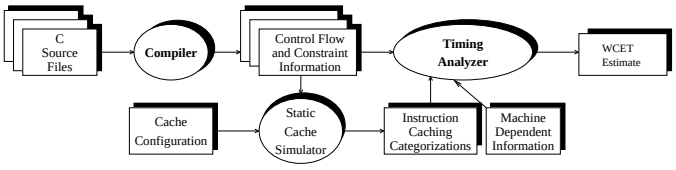


Fig. 1. Timing Analysis Tools

Throughout our work, we enhanced the timing analysis toolset in Figure 1 to determine not only the WCET but also the best case execution time (BCET) bounds, and not just for entire tasks but also for micro ranges of code. The original toolset provided timing feedback at the functional and loop level. We enhanced this capability to supply timing feedback for a series of smaller ranges within the same simulation run including aggregate values of WCET bounds for sequential instructions plus the cost of branch mispredictions. The resulting bounds are tight and enable us to determine, within a reasonable margin, if a security breach has occurred, *e.g.,* through attack code injection.

III. Design

This work puts forth the utilization of timing values readily available in real-time cyber physical systems to establish a intrusion detection technique. By utilizing our technique, critical and potentially vulnerable security-related information can be spread through-out the entire system. The primary goal of this work is to design and assess methodologies that provide real-time CPS applications with an intrusion detection security mechanism.

A. Timed Return Path Security (TRPS)

Timed Return Path Security (TRPS) is an application-level instrumentation that utilizes communication through the system clock in order to maintain a series of sanity checks structured throughout the code. To detect code injection attacks, we structure sanity statements mainly around application code that could potentially be overwritten, allowing the attacker to perform malicious actions. Such attempts are most commonly known as buffer overflow attacks. They involve overwriting the return address of a routine whose frames are stored on the stack. When the program executes the return statement of such a function. The control will be transferred to the location indicated by the overwritten return address. Attackers often choose a modified return value pointing into hand-written instructions. Such specialized attack codes may modify global program variables or even spawn new programs given sufficient knowledge of the affected application.

TRPS uses a mechanism to detect such attacks. Mischievous reasons for doing so may range from changing data for personal benefit to causing potentially catastrophic damage to the CPS environment, *e.g.,* to overload a power transformer by changing safety bounds data resulting in irreversible damage.

TRPS creates multiple sanity checks throughout an application at critical points where the program counter could potentially be transferred *via* a pointer to an undesignated area. These checks obtain clock information just before and after the return instruction as seen in steps 1 and 2 of Figure 2. Our method then utilizes the difference between the two time stamps and compares this delta against an already predetermined worst-case execution bound for the respective return path. This is depicted in step 3 of Figure 2. If the dynamically observed delta exceeds the WCET bound, excess instructions must be executed indicating a potential security compromise. In contrast to arbitrary code sections, static timing analysis on these focused regions yields tight WCET bounds since they mainly consist of a single straight-line execution path. Code sections subject to pointer-controlled flow transfers whose pointers are stored on stack generally comprise a series of loads and stores to restore prior processor state and unwind the stack. The communication structure of this method is displayed in Figure 2. It shows the application interfacing with the system twice to obtain values from the system clocks before checking the timestamp delta to validate WCET bounds. It is important to note that even if these regions exceed the measured WCET, it does not mean the overall program will exceed its calculated WCET. This makes TRPS well suited for detecting attacks that would not result in a deadline miss otherwise.

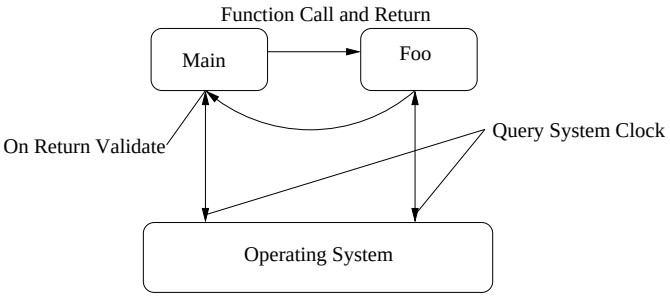


Fig. 2. Timed Return Path Security

IV. Implementation and Experimentation Framework

The overall framework for experimentation is depicted in Figure 3. We obtained our static WCET analysis tool that

provided us with the necessary timing analysis data [3], [6], [5]. The timing analysis tool was configured for a system utilizing the PISA instruction set. The cache configuration for both the static cache simulator and the timing analyzer were configured without data caches but with instruction cache misses accounted for in the WCET analysis. The choice of the cache configurations parameters was intentional as our objective here was to assess a bound on detectable code injections. In other words, given the tightest possible timings on application code, we wanted to determine the largest number of cycles that would remain undetected by our security-enhancing methods. For this metric, the smaller this threshold, the stronger is the protection.

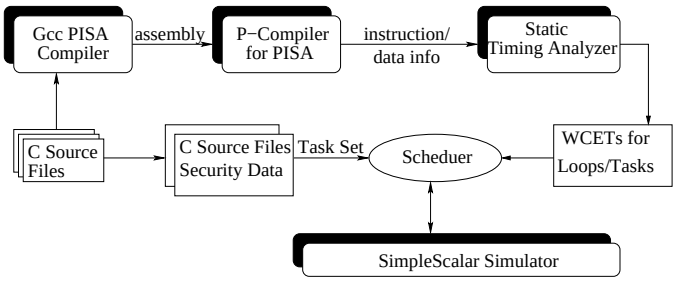


Fig. 3. Framework

To facilitate our experiments, we enhanced the timing analyzer with support for checkpointing instructions. These checkpointing instructions allow us to determine the exact cycle time at which a single instruction finishes execution.

We further obtained a customized version of the SimpleScalar processor simulator [1]. This modified version of SimpleScalar supports multitasking and has been enhanced to support a scheduler thread / task [4]. The target instruction set architecture for this simulator is PISA. This matches the input assembly utilized by our timing analysis tools. For the purpose of this work, we assess benchmark results in SimpleScalar configured with perfect branch prediction and prefect instruction caches but without data cache support. This matches the configurations of the static analysis tools.

As discussed before, these configurations provide a lower bound on the amount of code injection that may remain undetected. If we were to relax our configuration constraints, WCET bounds obtained by static analysis would become less tight implying that an attacker could potentially execute more instructions prior to being detected. Assessing such a trade-off is limited to a concrete implementation platform (see below) in this paper.

The scheduler utilized within the SimpleScalar framework supports multiple preemptive and non-preemptive scheduling algorithms. For the course of this work, we used a preemptive EDF schedule to most accurately show the side effect of our applied methods on the scheduler itself. Our implementation modified the scheduler to support relative time for each thread aggregated during preemptions and at security checks of a task to most accurately track the clock period of a particular task.

We further made the following enhancements to the SimpleScalar environment. We implemented two system calls to query timing information. Before a return from a function / method, the first system call is issued. At the destinations of a function / method return, the second system call is triggered. Both calls query the clock, and the difference in time between the two calls is then compared with static timing bounds for the respective code sections.

The motivation for creating two distinct system calls was to create a sequential ordering of these calls. If call one was issued without a corresponding call two (or vice versa), a control-flow violation is detected. Subsequently, a system-defined adverse action, such as transitioning into a fail-safe state, can be initiated. In effect, the imposed call ordering represents a security side-check that provides the means to detect certain attacks missed if only execution cycles were checked. For example, if an attacker were to execute injected code and then transfer control to the instructions past our second system call in an attempt to bypass our imposed security, the absence of the second system call would be detected at the next return from a function when another instance of the first system call is issued.

We tested our implementation using a set of floating-point and integer benchmarks from the C-Lab benchmark suite [2]. The actual benchmarks used are shown in Table I.

C Benchmark	Function
adpcm	Adaptive Differential Pulse Code Modulation
lms	An LMS adaptive signal enhancement
srt	Bubble Sort
fft	Fast Fourier Transform

TABLE I
C-LAB BENCHMARKS

V. RESULTS

In TRPS, the timed return path verification, utilizes an *absolute* task timer to determine the total time since the simulation start point.

Timed Return Path Security Results: Figure 4 depicts baseline / modified (TRPS) cycle overheads for WCET benchmarks SRT, LMS, ADPCM and FFT. The overheads, ranging between 0.22% and 18.71%, are often negligible or at most tolerable assuming sufficient slack in a real-time task schedule. Higher overhead in ADPCM is due to its modular structure compared to other benchmarks. It consists of several small functions that are called with a loop. Thus, our TRPS checks are invoked significantly more frequently (at nesting level one) than in other benchmarks (at nesting level zero — not inside of any loops).

Table II shows the sensitivity results of TRPS for various benchmarks and their respective functions. In this experiment, the attack code, after executing its injected code, returns to the exact spot in the code that the original return for a call would have jumped to. The table then reports the WCET in cycles for the return sequence as reported by timing analysis

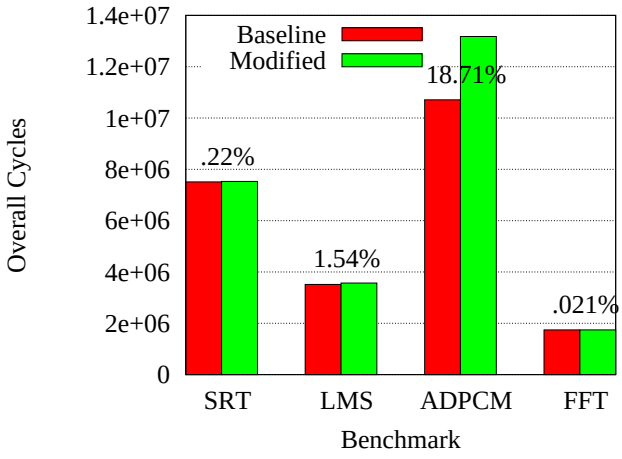


Fig. 4. TRPS Overhead

(column 3) and the number of slack cycles that would remain undetected (column 4). This slack amounts to the difference between WCET and actual execution time, the latter of which is observed from SimpleScalar simulation. The WCET bound is extremely tight since TRPS assesses time on a straight-line path of the control flow. Hence, the window of vulnerability is restricted to a sensitivity of 9-39 cycles. This limits the amount of code that may be injected code without being detected.

These results provide a lower bound, but it can be argued that the upper bound for undetectable injections is larger. First, an attacker could skip over selected instructions on the return path that manipulate registers and stack and instead inject their own code. However, disguising the side effects of polluting stacks and registers may not be trivial depending on the actual code. Conversely, we argue that additional security measurements are quite feasible, such as exploiting average case execution times for checks on timing outliers. Such methods are probabilistic and may result in large numbers of false positives. Nonetheless, early warning indicators could be dynamically triggered to activate stringent security checks that bare higher costs. Alternatively, system functionality could be reduced in order to limit potential damage to the *physical* side of the CPS application. Overall, the results in Table II illustrate that the timing estimations and subsequent security checks for straight-line code are very precise, thus leaving little room for injected code.

TABLE II
TRPS WCET AND SENSITIVITY 4KB I-CACHE[CYCLES]

Benchmark	Function	WCET	Sensitivity
SRT	Initialize	35	25
SRT	BubbleSort	45	19
LMS	LMS	28	18
FFT	FFT	25	8
ADPCM	Encode	93	11
ADPCM	Decode	65	39

VI. CONCLUSION

In this work, we developed a novel software methodology that provides enhanced security in deeply embedded real-time systems. We attain elevated security assurance through new levels of instrumentation that enable us to detect anomalies, such as timing dilations exceeding feasible bounds. We utilize tight timing bounds for selected code sections that are readily available at no extra cost whenever static timing analysis is required as part of schedulability analysis of a real-time system. The timing bounds are subsequently utilized to monitor execution during runtime. Upon validation of timing bounds, no action is taken. Upon violation of bounds, an alert is raised that provides an opportunity to reduce system functionality, revert to a fail-safe state or shut down the system altogether pending further investigation/assessment. To the best of our knowledge, such detection of system compromises through micro-timing information is a novel contribution to CPS environments with real-time constraints.

REFERENCES

[1] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Technical Report CS-TR-96-1308, University of Wisconsin - Madison, CS Dept., July 1996.

[2] C-Lab. Wcet benchmarks. Available from http://www.c-lab.de/home/en/download.html.

[3] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, Jan. 1999.

[4] S. Mohan, F. Mueller, W. Hawkins, M. Root, C. Healy, and D. Whalley. Parascale: Expoliting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *IEEE Real-Time Systems Symposium*, pages 233–242, Dec. 2005.

[5] S. Mohan, F. Mueller, D. Whalley, and C. Healy. Timing analysis for sensor network nodes of the atmega processor family. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 405–414, Mar. 2005.

[6] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

Real-Time Process Control in Producing Clean Air and Bio-Energy from Animal Waste

Yue Yu, Miao Song, Shangping Ren and Cindy Hood
Department of CS
Illinois Institute of Technology
Chicago, IL 60616
{yyu8, msong8, ren, hood}@iit.edu

Jun Zhu
Department of BBE
University of Minnesota
Waseca, MN 56093
zhuxx034@umn.edu

Gang Quan
Department of ECE
Florida International University
Miami, FL 33174
gang.quan@fiu.edu

***Abstract*—Rural areas offer a variety of waste bio-mass streams including animal wastes that can be used as non-food feedstock for bio-fuels and bio-energy production. Some processes that can produce electricity directly from waste bio-mass, such as from liquid swine manure, have been recently developed. However, a widespread application of bio-reactor technology at farm level has not ever been, and probably will never be, materialized unless new improvements in both the performance and economics of this technique are made. The major bottleneck rests with the poor efficiency resulted from little or no automated control of the bio-reactor system to assure the operations consistently under the optimal status. The focus of this WIP paper is to identify factors that affect the bio-reactor's operation status and build a event model to facilitate a formal understanding about the status of the system from lower and not always accurate physical data and making adaptive control decisions at application layer.**

I. Introduction

Bio-reactors are widely acknowledged to be an effective tool to treat animal manure in an environmentally friendly manner in that they can not only reduce the emissions of nuisance gases by completely blocking the transport pathway but also possess the capability of producing desired renewable products such as methane, bio-hydrogen, and organic chemicals. However, a widespread application of bio-reactor technology at farm level has not ever been, and probably will never be, materialized unless new improvements in both the performance and economics of this technique are made. The major bottleneck rests with the poor efficiency resulting from little or no automated control of the bio-reactor system so that it can be operated consistently in an optimal state.

Unfortunately, traditional real-time process control technologies used in closed plant environment face two challenging issues that prevent them from being readily used in the application mentioned above, in that they require to integrate communication, computation, and sensing and actuating through heterogeneous and widely distributed physical devices and require close interactions among disparate physical devices and computational components within and among physical world and cyber world. The focus of this WIP paper is to build a formal event model to facility non-ad-hoc integration between physical data and cyber process control layer.

The rest of the paper is organized as following: Section II describe the process of producing clean air and bio-energy from animal waste in details and identifies the issues to be solved relating to the automated interactions between different components of the system in a bio-energy production line. Section III presents an event model that semantically bridges the communications between the different physical devices of the system and lays a theoretic foundation for formally reasoning about the system. Related work and conclusions are made in Section IV, Section V, respectively.

II. The Process of Producing Clean Air and Bio-Energy from Animal Waste

The basic operating control of a bio-hydrogen system includes four subprocesses, that is influent tank process, the bio-reactor process where the main reaction happens, the PH controller process and the effluent process. Figure 1 depicts the setting for generating bio-hydrogen from liquid swine manure.

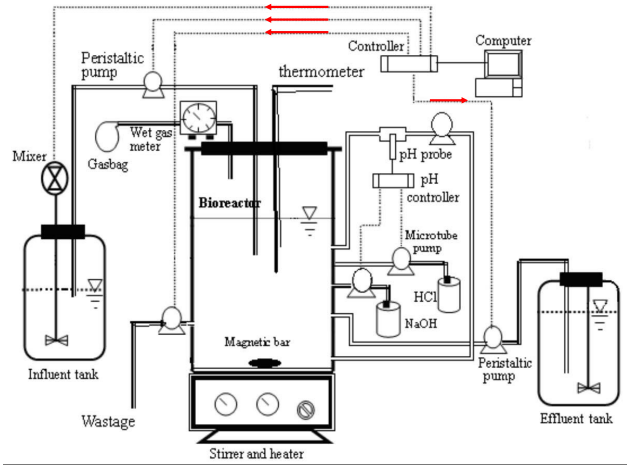


Fig. 1. Laboratory setting for generating bio-hydrogen from liquid swine manure

The influent tank provides the bio-reactor the manure liquid. The concentration of the manure liquid is controlled under a preset value within a $\pm$ error range. A mixer is used to ensure that the manure liquid has an even concentration within the influent tank. Only when the waste is well mixed before it can

be pumped from the influent tank to the bio-reactor. Clearly, with different waste concentration, the speed of the mixer as well as the time it takes to mix the waste liquid is different. We should adjust it accordingly as running mixer with higher speed or longer time than necessary increases the process cost.

The bio-reactor controls the factors that affect the operating status to assure the system running in the best productive condition. Four main factors are PH value, temperature, and concentration of the manure liquid, and reaction time. Signals are sent to the PH controller to add HCI or NaOH to the bio-reactor to maintain an optimal PH value which is based on the waste and the liquid concentration. Since the HCI or HaOH are injected through a single point, the PH value may be different from one point to the other within the reactor, a mixer is used in the reactor.

After a certain reaction time, the bio-gas are generated which is measured by a gas meter. As the chemical reaction proceeds, bio-gas generation speed peaks and then slows down until no bio-gas can be measured by the meter which indicates the completion of the bio-reaction. The waste water is then ejected into the effluent tank.

Clearly, the reaction time of the bio-reactor, the temperature, and the speed of the mixer all have impact to the total cost of the process and need to be well controlled and adjust at run-time. In order to assure the system running in its best productive condition, different types of data have to be gathered from different sections of the system's physical infrastructure and control decisions have to be made based on the local information.

Data generated from a component may affect the data operated in another different component. For instance, the speed of the influent tank mixer at one end may impact the speed of the micro-tube pump at the other end. To facilitate smooth interactions, events are often used to orchestrate the communication and interaction among different widely distributed and heterogeneous devices and components in the system. The challenge is how to uniformly represent events and be able to infer the "meaning" of an event that happens one place to the rest of the system.

III. Concept-Lattice-Based Event Model for CPS Systems

In order to fully represent a CPS event instance and be able to use it as a media to communicate between disparate cyber world and physical world, an event instance is defined below:

Definition 1 (CPS Event Instance): A CPS event instance has a type, a set of internal attributes and a set of external attributes. It is structured as in (1)

$$\mathcal{E}_{cps} : \Gamma\, \mu\, @\, (\mathcal{T},\, \mathcal{L},\, \mathcal{O}) \tag{1}$$

where,

- Γ represents the type of the event instance;
- μ represents the internal attribute of the instance;
- $\mathcal{T}$ indicates the time when the event instance happens.
- $\mathcal{L}$ indicates the location where the event instance occurs.
- $\mathcal{O}$ is an observer of the event instance. The existence of an observer is also treated as an event.
- $\mathcal{O}_\top$ is the global observer of a CPS system. Its location is the system's origin, and its time interval is defined as the system's life span, i.e., $\mathcal{O}_\top = Object\ \top\ @\ ([0,\infty),\ ((0,0,0),\infty),\ \top)$. There is one and only one global observer $\mathcal{O}_\top$.

□

Also, We define the value function $\mathcal{V}(\mathcal{E}_{cps})$, time function $\mathcal{T}_o(\mathcal{E}_{cps})$, location function $\mathcal{L}_o(\mathcal{E}_{cps})$, observer function $\mathcal{B}(\mathcal{E}_{cps})$ to extract the corresponding information of Γ and μ, $\mathcal{T}$, $\mathcal{L}$, $\mathcal{O}$, respectively.

To abstract from lower physical level events to the ones with richer meanings toward cyber world, we adopt the theory of concept lattice [1].

A. Concept Lattice

The theory of concept lattice is established upon a *formal context*. In our case, a formal context is defined as:

Definition 2 (Formal Context): a formal context is a triple $(\mathbb{I}, \mathbb{E}, R)$, where $\mathbb{I}$ and $\mathbb{E}$ are internal attribute and type sets of events. $R \subseteq \mathbb{I} \times \mathbb{E}$ is a binary relation where $(\mu, \Gamma) \in R$ denotes that event instant with internal attributes μ has type Γ.

□

A formal context defines the relationship between event internal attributes and basic event types. For example, if we classify the concentration of liquid level as `low`, `medium`, and `high`, and their temperature as `cool`, `warm`, and `hot`, the formal context can be defined as $(\langle C(oncentration), T(emperature)\rangle, \{c_l, c_m, c_h, t_c, t_w, t_h\}, R)$, where R is defined as the set

$$\left\{ \begin{array}{c} (\langle [0\%, 30\%), T\rangle, h_l), (\langle [30\%, 60\%), T\rangle, h_m), \\ (\langle [60\%, 100\%], T\rangle, h_h), (\langle H, [32F, 86F)\rangle, t_c), \\ (\langle H, [86F, 140F)\rangle, t_w), (\langle H, [140F, 212F]\rangle, t_h) \end{array} \right\}$$

Relations between event attributes and basic types in a formal context are composed to establish the relationships between more abstract event internal attributes and more complicated event types. These relationships are called called *concepts* defined below:

Definition 3 (Concept): Let $(\mathbb{I}, \mathbb{E}, R)$ be a formal context, (X, Y) where $X \in 2^{\mathbb{I}}$, $Y \in 2^{\mathbb{E}}$, is called a *concept*, if

$$X = \{\mu \in \mathbb{I} | \forall \Gamma \in \mathbb{E}, (\mu, \Gamma) \in R\} \tag{2}$$

$$Y = \{\Gamma \in \mathbb{E} | \forall \mu \in \mathbb{I}, (\mu, \Gamma) \in R\} \tag{3}$$

□

For example, liquid with `medium` concentration level and `warm` temperature, i.e., $\langle [30\%, 60\%), T\rangle \cap \langle H, [86F, 140F)\rangle = \langle [30\%, 60\%), [86F, 140F)\rangle$ is of type $\{c_m, t_w\}$. We associate event instance $\langle [30\%, 60\%), [86F, 140F)\rangle$ with type $PerfectCondition$ (an alias for $\{c_m, w_m\}$) as a concept.

A formal context and its concepts establish the relationships between event types and their corresponding valid internal attributes. The next step is to build a hierarchy of such relationships in order to lay the semantic base for event

abstractions. Such a hierarchy is defined as a *concept lattice* given in the following definition:

Definition 4 (Concept Lattice): For a formal context $(\mathbb{I}, \mathbb{E}, R)$, let (X_1, Y_1) and (X_2, Y_2) be two concepts. If $X_1 \subseteq X_2$, or $Y_1 \supseteq Y_2$, then there is a partial order $\prec$ between (X_1, Y_1) and (X_2, Y_2), i.e.,

$$(X_1, Y_1) \prec (X_2, Y_2) \tag{4}$$

Such an partial order relation is of a lattice structure and forms the *concept lattice* of the formal context $(\mathbb{I}, \mathbb{E}, R)$.

□

With a concept lattice, we are able to obtain event instances with richer meanings from event instances that contain primitive data, but with less type information.

B. Event Type and Internal Attributes Composition

Given a concept lattice, a set of event instances can be composed if and only if their corresponding event type and internal attributes are composable in the concept lattice. Further the composed type and internal attribute values (or the composed concept) must be the largest lower bound of the composing concept. Definition 5 gives the formal definition.

Definition 5 (Event Γ and μ Composition ($\mathcal{A}_\mathcal{V}$)): Given a set of CPS events $e_1, e_2, \cdots, e_n$, with their types and internal attributes as $\Gamma_1\,\mu_1,\ \Gamma_2\,\mu_2,\ \cdots,\ \Gamma_n\,\mu_n$, respectively, and a concept lattice $\mathbb{C}$

$$\mathcal{A}_\mathcal{V}(\Gamma_1\,\mu_1, \Gamma_2\,\mu_2, \cdots, \Gamma_n\,\mu_n) = \Gamma\ \mu$$

If $\Gamma_1\ \mu_1$, $\Gamma_2\ \mu_2$, ..., $\Gamma_n\ \mu_n$ are all immediate successors of $\Gamma\ \mu$ in the given lattice $\mathbb{C}$.

□

The concept lattice only captures the event type and event instance's internal attributes. However, event instance also has a non-separable external attributes, i.e., its occurrence time and location and its observer. Certain event compositions may only be permissible under certain conditions which involve the event instances external attributes, we call such event compositions as *guarded compositions*.

Guarded compositions are integrated into the concept lattice by labeling the edges with the corresponding event constraints.

For example, consider two events: liquid with concentration 35% filled in a tank $\Gamma_1\,\mu_1 = Medium\,\langle 35\%, -\rangle$ and the temperature of liquid in the tank is 142F, $\Gamma_2\,\mu_2 = Gethot\,\langle -, 142F\rangle$ where the internal attributes of event instances are of the form $\langle C(oncentration), T(emperature)\rangle$, "$-$" denotes the don't-care attribute. The composition of the two events are defined as

$$\Gamma\,\mu = MediumGethot\,\langle 35\%, 142F\rangle$$

where $MediumGethot$ is a shorthand notation for the set $\{Medium, Gethot\}$. As can be easily checked, for internal attributes of events, we have $\langle 35\%, T\rangle \supseteq \langle 35\%, 142F\rangle$ and $\langle C, 142F\rangle \supseteq \langle 35\%, 142F\rangle$; and for data types, we have $\{Medium\}, \{Gethot\} \subseteq \{Medium, Gethot\}$. Therefore, according to Definition 4, we have $\Gamma\ \mu \prec \Gamma_1\ \mu_1$ and $\Gamma\ \mu \prec \Gamma_2\ \mu_2$ in the concept lattice, graphically denoted as:

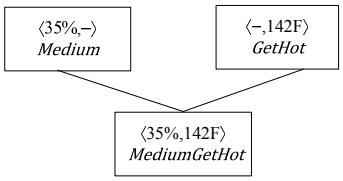


Fig. 2. An Example of Concept Lattice

Clearly, an event instance of type $MediumGethot$ has richer meaning than an event instance of type $Medium$, or $Gethot$.

Event compositions based on a given concept lattice allows events to be transferred cross different boundaries and understood by heterogeneous devices and components distributed in a CPS system.

The bottom of a concept lattice is defined by the global observer of the system $\mathcal{O}_\top$.

We apply the concepts and notations to define events and their relationships in the application of producing clean air and bio-energy from animal waste and show how we obtain different layers of abstractions from physical data.

Example 1: As shown in Fig. 1, the mixer in fluent tank agitate the swine manure and some other supplements together providing H_2-producing bacteria to the bioreactor. Two sensors which are installed at (3,5,0) and (1,2,0) on the bottom of the tank observe the concentration of the mixture, and their sensing range are of radius 4 and 3, respectively. In addition, there is another sensor installed at (2,4,12) sensing the level change of the mixture with the radius of 13. We want to describe a scenario that when the concentration of the mixture in the tank is of uniformity, we need to slow down the speed of mixer and simultaneously, the pump begin to feed the mixture to the reactor. When the level of the mixture decreases 5 meters, we need to stop feeding and new mixtures are refilled. The span of the fluent tank is (0,0,0),15). Assume the span of the fluent tank is (0,0,0),15),and the level of mixture is up to 11m.

□

For this example, there are nine event types. As we are only concerned with the concentration and the level attributes of event instance, the form $\langle C(oncentration), H(eight)\rangle$ is used to express the set of attributes. Event composition is to compose basic and simple events into more abstract events. More specifically, in this example, there is a `Sensor` event type which observes the `Height` event type and the `Concentration` event type, the `Height` event type can be composed into a new `HeightDecreaseby5m` event type if some constraint conditions are satisfied. Similarly, the `Concentration` event type can be composed into the `UniConcentraion` event type, the latter can be further composed into the `Slow-down` event type and `Beginfeed` event. When the `HeightDecreaseby5m` type event occurs after the `Beginfeed` event, we can compose the two into the new `Stopfeed` event, based on which the new `Refill` mixture event is generated.

The concept lattice for the example is given in Fig. 3.

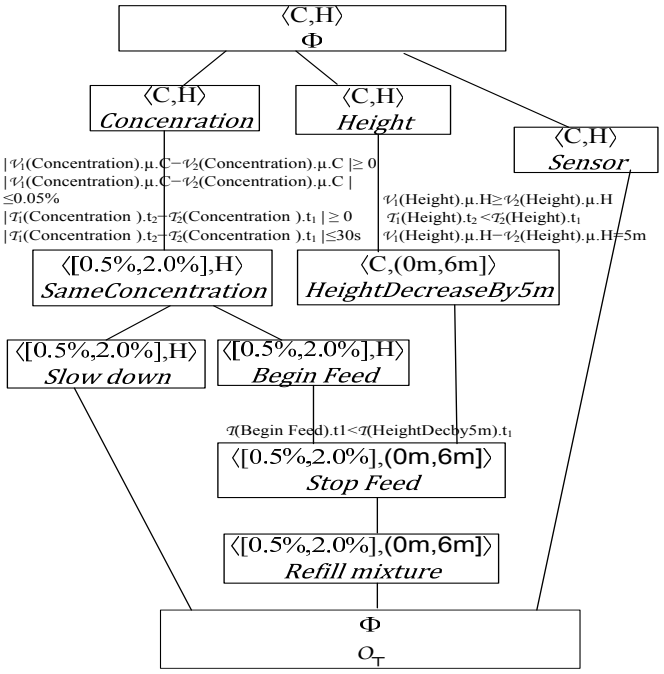


Fig. 3. Concept Lattice for xxx

IV. Related Work

The concept of events has been investigated in many different contexts both in the cyber domain and the physical domain. The *event calculus* [2], [3], [4] investigates a logic program framework for representing and reasoning about events (or *actions*) and their effects. Under this framework, time-varying properties (true or false) of the world during certain intervals, called *fluents*, are initiated by an occurrence of an action continue to (or not to) hold until an occurrence of an action which terminates them.

In [5], the concept of *observers* and a hierarchical spatial-temporal event model for CPS are introduced. The event model uses event attributes, occurrence time and space stamps, and event observer together to uniquely identify a CPS event instance. In addition, a set of temporal, spatial and logical operators are defined to support the temporal and spatial event composition. However, in [5], events are differentiated by four categories based on the corresponding four different system layers, namely, *physical events*, *sensor events*, *cyber-physical events*, and *cyber events*. Furthermore, although [5] defines event temporal, spatial, and logical compositions, structural representation of observers, and formal treatment of event type compositions are missing from the work.

The theory of *concept lattice* [1], [6] (Galois lattice) is a conceptual hierarchical structure based on binary relation proposed by Rudolf Wille [1], [7]. The theory has been widely used in the fields of software engineering [8], [9], [10] and data mining [11], [12], [13]. To the best of our knowledge, it is the first attempt to use *concept lattice* as an interpretation media transfer events across heterogeneous boundaries.

V. Conclusion

This paper presents a real-time control process system in producing clean air and bio-energy from animal waste. We further discover that to successfully build the expressions of different communications in the system, it is critical to have a unified representation for events at different abstraction layers and have a formal event transformation mechanism to relay events across different layers and device boundaries. We have made the first attempt to apply the theory of *concept lattice* in defining the relationship among event types within a heterogeneous and widely distributed system. Our initial excises seems indicating that the unified event representation and *concept lattice* can represent most of the scenarios that we have encountered. Our next step is to first prototype a system with real sensors installed in the lab settings, build the concept lattice for automatic event transformation and hence controlling the whole system.

Acknowledgment

This work is supported in part by NSF under grants CNS-0746643, CNS-0545913, and CNS-0917021.

References

[1] R. Wille, "Formal concept analysis as mathematical theory of concepts and concept hierarchies," *Lecture Notes in Computer Science*, vol. 3626, pp. 1–33, 2005.

[2] R. Kowalski and M. Sergot, "A logic-based calculus of events," *New Gen. Comput.*, vol. 4, no. 1, pp. 67–95, 1986.

[3] E. T. Mueller, "Automating commonsense reasoning using the event calculus," *Commun. ACM*, vol. 52, no. 1, pp. 113–117, 2009.

[4] V. Ermolayev, N. Keberle, and W.-E. Matzke, "An ontology of environments, events, and happenings," in *Proceedings of the 32nd Annual IEEE International Computer Software and Applications, COMPSA'08*, 2008.

[5] Y. Tan, M. C. Vuran, and S. Goddard, "Spatio-temporal event model for cyber-physical systems," *Distributed Computing Systems Workshops, International Conference on*, vol. 0, pp. 44–50, 2009.

[6] M. Liquiere and J. Sallantin, "Structural machine learning with galois lattice and graphs," in *Proc. of the 1998 Int. Conf. on Machine Learning*, 1998, pp. 305–313.

[7] R. Wille, "Restructuring lattice theory: An approach based on hierarchies of concepts," *Ordered Sets, Ivan Rival Ed., NATO Advanced Study Institute*, vol. 83, pp. 445–470, Sep. 1981.

[8] C. Lindig and A. Softwaretechnologie, "Concept-based component retrieval," in *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, 1995, pp. 21–25.

[9] G. Snelting, "Reengineering of configurations based on mathematical concept analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 146–189, 1996.

[10] G. Snelting and F. Tip, "Reengineering class hierarchies using concept analysis," in *In ACM Trans. Programming Languages and Systems*, 1998, pp. 99–110.

[11] B. Groh and P. W. Eklund, "Algorithms for creating relational power context families from conceptual graphs," in *Proceedings of the 7th International Conference on Conceptual Structures*, 1999, pp. 389–400.

[12] R. Cole and P. Eklund, "Scalability in formal concept analysis," *Computational Intelligence*, vol. 15, pp. 11–27, 1999.

[13] R. Cole, P. Eklund, and G. Stumme, "CEM – a program for visualization and discovery in email," in *Proc. Principles of Data Mining and Knowledge Discovery.*, vol. 1910, 2000, pp. 367–374.