Exploiting VISA for Higher Concurrency in Safe Real-Time Systems

Aravindh Anantaraman, Kiran Seth, Eric Rotenberg, Frank Mueller

ECE and CSC Departments,
Center for Embedded Systems Research, North Carolina State University

{avananta, krseth, ericro}@ece.ncsu.edu, mueller@cs.ncsu.edu

Abstract

Worst-case execution times (WCET) of tasks are essential for safe scheduling in hard real-time systems. However, contemporary processors exceed the capabilities of static worst-case timing analysis tools. The Virtual Simple Architecture (VISA) framework shifts the burden of bounding the WCET of tasks, in part, to hardware. A VISA is the pipeline timing specification of a hypothetical simple processor. WCET is derived for a task assuming the VISA. Nonetheless, at run-time, the task is executed speculatively on an unsafe complex processor, and its progress is continuously gauged. If continued safe progress appears to be in jeopardy, the complex processor is reconfigured to a simple mode of operation that directly implements the VISA, thereby explicitly bounding the task's overall execution time by the WCET.

In practice, the complex processor finishes tasks much faster than an explicitly-safe simple processor, creating significant slack in the task schedule. In previous work, this slack was exploited to safely lower frequency/voltage for power savings, in systems with only periodic hard-real-time tasks. In mixed systems with periodic and sporadic hard-real-time tasks, as well as soft-real-time tasks, the slack can be exploited for higher throughput and, hence, increased functionality/quality-of-service. This paper explores the throughput benefits enabled by the VISA framework, using both single-threaded and simultaneous multithreading processors. Using 10 tasksets composed from the C-lab benchmark suite (for periodic and sporadic hard-real-time tasks) and MPEG (for soft-real-time tasks), we show that 2.4-16 times more sporadic tasks are accepted using a VISA-protected complex processor versus an explicitly-safe simple processor. No MPEG frames are dropped by the complex processor whereas 67-93% of frames are dropped by the simple processor. Another contribution is demonstrating the feasibility of the

VISA framework in mixed (hard/soft, periodic/sporadic) multi-tasking systems and its transparency to the task scheduler.

1 Introduction

A hard-real-time embedded system is characterized by a collection of tasks (called a taskset), where each task executes repeatedly with some pre-specified rate. We refer to such tasks as *periodic hard-real-time tasks*. The *period* of a task is the time between successive invocations. Schedulability analysis is done beforehand to determine whether or not all of the tasks can be executed concurrently on the processor, always adhering to the periodicity of each task. Since tasks repeat indefinitely and at different rates, schedulability analysis is based on a utilization test. Utilization of a task is the fraction of time that the processor spends executing the task. In the worst case, the utilization of a task *i* is simply its worst-case execution time (WCET) divided by its period: $U_i = WCET_i / period_i$. The example in Figure 1 shows two tasks A and B with worst-case utilizations of 1/3 ($WCET_A$ = 1 unit, $period_A$ = 3 units) and 1/4 ($WCET_B$ = 1 unit, $period_B$ = 4 units), respectively. Task *release* points indicate when the next instance of a task is queued by the scheduler and occur at constant intervals equal to the period. The total utilization of a taskset is the sum of individual task utilizations, and must be less than 1 for the taskset to be schedulable under worst-case conditions, assuming earliest-deadline-first (EDF) scheduling [15]. The taskset in Figure 1 is schedulable since its total utilization is 7/12 < 1.

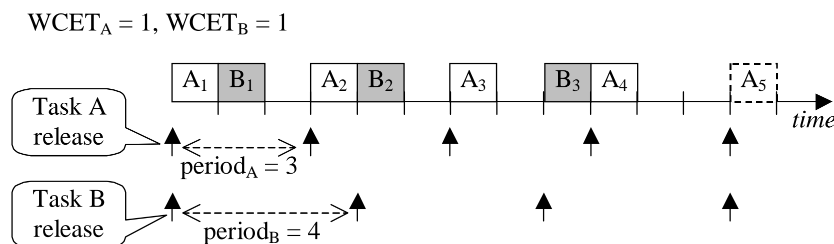

Figure 1. Example showing two periodic hard-real-time tasks.

Hence, WCETs of tasks are essential for safe scheduling in hard real-time systems since they are used to compute worst-case utilization. For this reason, there has been much research geared towards static worst-case timing analysis tools [2][7][8][9][11][12][13][14][16][18][19][20][23][25]. These tools are used to automatically derive safe and tight WCET bounds of tasks independent of input data and taking into consideration the cycle-accurate timing of the microarchitecture. Much of the research

revolves around uncertainty introduced by variable control flow and data flow as well as microarchitectural uncertainty (e.g., caching effects).

Deriving safe WCET bounds for tasks executing on contemporary processors is a difficult problem. Currently, static worst-case timing analysis tools are limited to scalar in-order pipelines with split caches and static branch prediction. The use of complex processors in hard real-time systems is either barred, or complex features are disabled for the duration of hard-real-time tasks [6].

The Virtual Simple Architecture (VISA) framework was recently introduced to enable the unrestricted use of complex processors in safe real-time systems [1]. A VISA is the pipeline timing specification for a hypothetical simple processor, which is within the capabilities of static worst-case timing analysis. The WCET of a task is derived assuming that the task will execute on the VISA. This assumption is undermined at run-time since the task is actually executed on a complex pipeline. Since the WCET was not derived for the complex pipeline explicitly, a means is required for dynamically confirming that the task's execution time is in fact bounded by the WCET. This is achieved by dividing the task into multiple smaller sub-tasks. Sub-tasks provide a means to gauge progress on the complex pipeline. Each sub-task is assigned a soft interim deadline, called a checkpoint, which is based on the latest allowable completion time of the sub-task assuming it was executing on the VISA. Accordingly, if a sub-task completes before its checkpoint, then continued safe progress is confirmed. If a sub-task misses its checkpoint, then the complex pipeline is reconfigured to a simple mode of operation that directly implements the VISA specification (e.g., out-of-order execution is disabled, the fetch unit reverts to static branch prediction, simultaneous multithreading is disabled, etc.). The unfinished sub-task and all remaining sub-tasks are executed in the simple mode, ensuring that the WCET bound is not exceeded in spite of the missed checkpoint.

In practice, periodic hard-real-time tasks finish much faster on the complex processor than on an explicitly-safe simple processor. Exceeding performance requirements does not benefit the periodic hard-real-time tasks themselves – they are schedulable in any case. Moreover, exceeding performance requirements does not enable more periodic hard-real-time tasks to be scheduled because the *worst-case* utilization is not reduced with respect to a simple processor – the WCET abstraction enforced by the

VISA framework is based on the hypothetical simple processor. Nonetheless, the *actual* utilization is dramatically reduced compared to a simple processor. That is, significant dynamic slack is created in the schedule and, although dynamic slack cannot be applied to additional periodic hard-real-time tasks, it can be used to accommodate other kinds of tasks or applications, namely the following.

- *Sporadic hard-real-time tasks*. These are hard-real-time tasks that request execution irregularly. The scheduler is invoked when a sporadic task is released (i.e., queued), but the scheduler only schedules the sporadic task if it can be fit into the schedule (based on a dynamic worst-case utilization test). This is called an acceptance test. Sporadic hard-real-time tasks differ from periodic hard-real-time tasks in that it is safe to deny them but unsafe to accept them without first ensuring their timely completion.
- *Periodic soft-real-time tasks*. These are periodic tasks (like periodic hard-real-time tasks) but it is safe to "drop" instances of a task. Periodic soft-real-time tasks are not included in the worst-case utilization and, as a result, they do not hold the same status as periodic hard-real-time tasks in terms of guaranteeing a certain rate of execution. However, more generally, a certain quality-of-service (QoS) level can be guaranteed by including the periodic soft-real-time task at a reduced rate in the worst-case utilization.
- *Conventional applications*: These are non-real-time applications.

The VISA framework enables the use of complex processors while preserving the WCET abstraction of hard-real-time tasks, opening up opportunities for safely exploiting higher performance. Recent previous work with respect to VISA exploits higher performance for safely lowering frequency/voltage of periodic hard-real-time tasks, both in single-task [1] and multi-tasking systems [21]. However, this previous work does not consider mixed-task systems composed of periodic hard-real-time tasks, sporadic hard-real-time tasks, periodic soft-real-time tasks, and general applications. The primary contribution of this paper is exploring the throughput advantages enabled by the VISA framework. Using 10 tasksets composed from the C-lab benchmark suite (for periodic and sporadic hard-real-time tasks) and an MPEG decoder (for soft-real-time tasks), we show that 2.4-16 times more sporadic tasks are accepted

using a VISA-protected complex processor versus an explicitly-safe simple processor. No MPEG frames are dropped by the complex processor whereas 67-93% of frames are dropped by the simple processor.

Another contribution of this paper is demonstrating the feasibility of the VISA framework in mixed-task (hard/soft, periodic/sporadic) multi-tasking systems, and demonstrating the transparency of the VISA framework to the task scheduler in these systems. The VISA framework preserves the WCET abstraction, which is the basis for schedulability analysis, task scheduling (e.g., earliest-deadline-first or EDF scheduling), and all other software layers within real-time systems. As such, the VISA framework is transparent to the task scheduler. For example, we implemented a standard EDF scheduler for periodic hard-real-time tasks, augmented with a standard acceptance test for sporadic hard-real-time tasks [15]. Neither aspect of the scheduler is impacted by the VISA framework since the analyses are based solely on the WCETs and deadlines of tasks.

The remainder of this paper is organized as follows. Related work is described in Section 2. Section 3 discusses how a VISA-compliant complex processor can be exploited for higher concurrency, including an overview of task scheduling. Section 4 describes the means by which the VISA framework guarantees safety of hard-real-time tasks on otherwise unsafe processors. The experimental methodology is described in Section 5. Experimental results are presented in Section 6. Section 7 summarizes the paper and discusses future work.

2 Related Work

Scheduling theory for hard real-time systems, such as rate-monotone and EDF scheduling, relies on known WCET bounds for tasks [4][15][22]. For modern architectures, timing bounds on a task's WCET are increasingly difficult to obtain as the complexity of architectures increases [6]. Past research on bounding WCETs of real-time programs based on static timing analysis has evolved from unoptimized programs on simple CISC processors [7][19][20] to optimized programs on pipelined RISC processors [8][14][25], and from uncached architectures to instruction caches [2][9][12][18] and data caches [11][13]. Analysis techniques of these approaches range from data-flow analysis [2][18], integer-linear programming [12], abstract interpretation [23], and path enumeration [20] to architectural simulation [16].

While the capabilities of static worst-case timing analysis will undoubtedly continue to improve, the VISA framework expedites the introduction of contemporary (unsafe) processors into hard real-time systems, by decoupling worst-case timing analysis from the underlying microarchitecture [1]. Initial VISA research exploits the resulting decrease in actual utilization to safely lower frequency/voltage, reducing energy consumption with respect to non-VISA systems. Energy reduction was observed in both single-task [1] and multiple-task systems [21]. This prior work only considers periodic hard-real-time tasks and, as such, cannot exploit VISA for higher concurrency. In contrast, this paper considers mixed-task systems (hard/soft, periodic/sporadic), which are able to exploit the concurrency opportunities exposed by the VISA framework.

Jain, Hughes, and Adve provide the first (and extensive) study of soft real-time scheduling on SMT processors [10], pointing out the unique opportunities and challenges in this new setting. They divide the problem into two sub-problems, co-scheduling (which tasks to run simultaneously) and resource sharing (how to share resources among co-scheduled tasks). Key factors in achieving schedulability include prioritization of high-utilization tasks and exploitation of symbiosis. They only consider soft-real-time tasks and consequently consider a taskset to be schedulable even if some fraction (5%) of deadlines are missed. In contrast, we consider mixed systems composed of both hard-real-time and soft-real-time tasks, where the novelty is in improving throughput of soft-real-time tasks by speculatively undermining the safety of hard-real-time tasks, and ultimately guaranteeing the safe completion of the latter via the VISA safety mechanisms. Our work and the work by Jain *et al.* are complementary – although beyond the scope of this paper, in the future, we would like to integrate their soft-real-time SMT scheduling techniques within a VISA framework.

3 Exploiting the VISA Framework for Higher Concurrency

In Section 3.1, we use an example to compare and contrast execution on an explicitly-safe simple processor versus a VISA-compliant complex processor. The example will show that speculation increases worst-case utilization of periodic hard-real-time tasks, but the reduction in actual utilization far outweighs this overhead.

Section 3.2 describes the scheduler. The scheduler implements EDF scheduling for periodic hard-real-time tasks, an acceptance test for sporadic hard-real-time tasks, and background scheduling of periodic soft-real-time tasks and non-real-time applications.

3.1 Creating slack

Figure 2a shows a task executing on an explicitly-safe simple processor. There are four sub-tasks. (Sub-tasks are not needed in this case because the simple processor is safe, but they are shown for comparison later.) Dashed vertical lines indicate the WCETs of sub-tasks. Arrows indicate the actual execution times of sub-tasks. In the example, actual execution times are close to WCETs.

Figure 2b-c shows the speculative approach, i.e., execution on a VISA-compliant complex processor. Dashed vertical lines indicate checkpoints. Notice that the overall WCET of the task is extended (padded) to enable speculation, and the checkpoint for each sub-task lines up with the start time of the sub-task in the case of non-speculative execution (Figure 2a). Padding is needed so that there is enough time to complete an unfinished sub-task and remaining sub-tasks (assuming worst-case conditions) if a checkpoint is missed. This increases worst-case utilization with respect to the non-speculative approach. However, as will be seen, this overhead is far outweighed by the dramatic reduction in actual utilization, resulting in significant dynamic slack that can be exploited for lower power [1][21] or higher concurrency, the latter being the focus of this paper.

The amount of WCET padding is arbitrary, although the more padding, the less likely a checkpoint will be missed and the higher the overhead of speculation. In Section 4.3, we give a rationale for padding with the maximum WCET among all sub-tasks of the task. Hence, increasing the number of sub-tasks is one approach for reducing the overhead.

Figure 2b shows the speculative approach in which no checkpoints are missed. Sub-tasks complete well before their checkpoints. This is what we expect virtually all the time. Dynamic checking and the fall-back simple mode are only needed because we cannot statically prove checkpoints will be met.

Figure 2c shows the speculative approach in which a checkpoint is missed. We say sub-task 2 is mispredicted because it does not complete before its checkpoint. Missed checkpoints are detected via a watchdog timer (described later in Section 4.2). When a checkpoint is missed, the overall task's execution time cannot be safely bounded if execution continues on the complex processor. The complex processor is reconfigured to operate in the simple mode so that the pipeline timing matches the VISA for which WCET was derived. In Figure 2c, execution in the simple mode is shown with dashed arrows instead of solid arrows. We cannot determine how much of the mispredicted sub-task (sub-task 2) completed before the checkpoint. We must be conservative and assume that none of sub-task 2 was completed, hence all of sub-task 2 must execute in the simple mode. This is why the checkpoint of a sub-task lines up with the start time of the sub-task in the non-speculative case – enough time is allotted to execute the entire mispredicted sub-task along with all remaining sub-tasks assuming worst-case conditions. Notice that the sub-tasks executed in the simple mode (Figure 2c, dashed arrows) line up with the corresponding sub-tasks in the non-speculative case (Figure 2a).

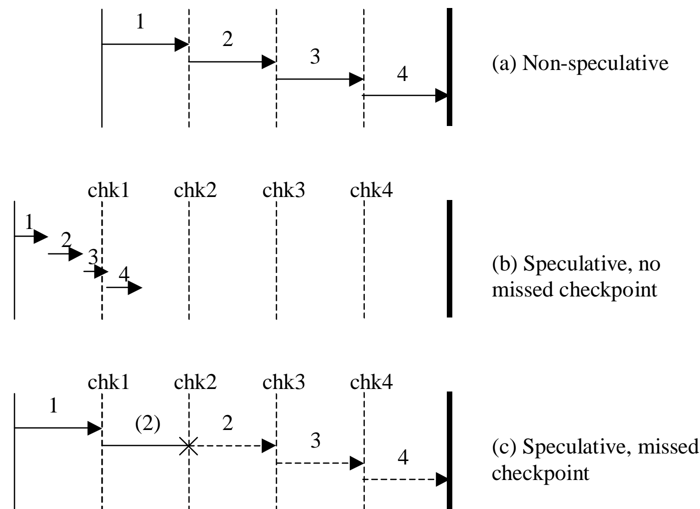

Figure 2. (a) Conventional (non-speculative) execution on an explicitly-safe simple processor. (b) Speculative approach, no checkpoints are missed. (c) Speculative approach, a checkpoint is missed (sub-task 2).

3.2 Scheduler

Schedulability analysis is done during system development to determine if the taskset is schedulable. It primarily involves the utilization test described earlier. Schedulability analysis should not

be confused with actual scheduling of tasks. Very often, scheduling is deferred until run-time and is performed by the task scheduler.

We use a standard earliest-deadline-first (EDF) scheduler in this paper. We assume the deadline of a task is its period (i.e., once released, a task must finish before the next release). An example of EDF scheduling is shown in Figure 3. Task A and B are released at the same time. Task A is scheduled first because it has the earlier deadline. Task B is scheduled when Task A completes. While Task B is executing, Task A is released for a second time (according to its period). Task B is preempted and Task A is scheduled, because the second instance of Task A has an earlier deadline than Task B. Preemptions are quite common in real-time systems. In Section 4.4, we discuss considerations for preemptions within the VISA framework.

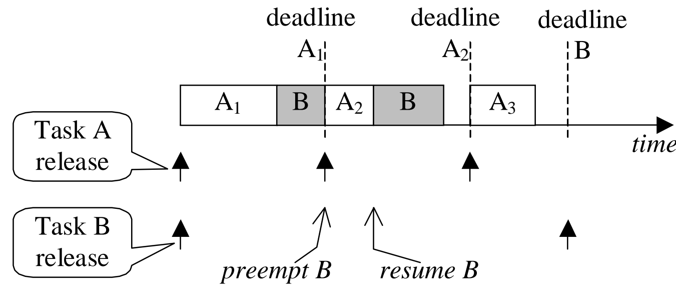

Figure 3. Example of EDF scheduling.

The scheduler also performs an acceptance test when a sporadic hard-real-time task is released. The scheduler makes a decision on whether or not to accept the sporadic task, based on the worst-case utilization between the current time and the deadline of the sporadic task [15]. This dynamic utilization test must include (1) the residual WCET of the current task, (2) the residual WCETs of any preempted tasks, (3) the WCETs of any already-released tasks, (4) the WCETs of any tasks that will be released within the considered timeframe, and (5) the WCET of the sporadic task itself. If the calculated worst-case utilization is less than or equal to 1, then the sporadic task is accepted, otherwise it is denied.

Finally, the scheduler also schedules periodic soft-real-time tasks. When a soft-real-time task is released, the scheduler checks to see if the previous instance of the task has completed. If the previous instance has not yet completed, then the current instance is dropped, degrading the quality-of-service (QoS) for the periodic soft-real-time task.

The scheduler is more likely to accept sporadic hard-real-time tasks and achieve high QoS for periodic soft-real-time tasks if there is significant dynamic slack. Thus, the VISA-compliant complex processor has an advantage over an explicitly-safe simple processor.

4 VISA Safety Framework

This section describes the steps taken to ensure safe operation on a VISA-compliant complex processor.

4.1 Setting checkpoints

As described earlier in Section 3.1, if a sub-task does not complete before its checkpoint, there must be enough time between the checkpoint and the deadline to execute the entire mispredicted sub-task and all remaining sub-tasks. We cannot determine how much of the mispredicted sub-task was executed before the checkpoint, so we assume none of it was executed. In addition, pipeline switching overhead must be budgeted. The overhead includes the maximum time to drain or squash (the choice is implementation-dependent) instructions from the pipeline before switching to the simple mode, plus the time to reconfigure the pipeline for the simple mode of operation. Assuming there are s total sub-tasks in the task, the checkpoint for a sub-task i is given by the expression below. The sum term corresponds to the combined worst-case execution time of the mispredicted sub-task i and all remaining sub-tasks.

$$\text{checkpoint}_i = \text{deadline} - \text{overhead} - \sum_{k=i}^{s} \text{WCET}_k$$

4.2 Enforcing checkpoints via the watchdog counter

A watchdog counter facilitates detection of missed checkpoints. A code snippet at the start of the first sub-task initializes the watchdog counter to the number of cycles between the start of the task and the first checkpoint, or $\lceil \text{checkpoint}_1 * \text{freq} \rceil$ (where freq is the processor frequency). A code snippet at the beginning of each new sub-task augments the counter by the number of cycles between the previous checkpoint and the sub-task's checkpoint, or $\lceil (\text{checkpoint}_i - \text{checkpoint}_{i-1}) * \text{freq} \rceil$, effectively advancing the checkpoint enforced by the watchdog counter. Hardware decrements the counter by one each cycle. If the counter reaches zero before the next sub-task has a chance to advance the counter, it means the current sub-task missed its checkpoint and a missed-checkpoint exception is raised. This causes the complex

processor to switch to the simple mode of operation. Missed-checkpoint exceptions are masked while the complex processor is operating in the simple mode, since progress does not need to be gauged while in the simple mode.

4.3 Padding the WCET

As described earlier in Section 3.1, the WCET provided to the scheduler must be padded to enable speculation. There is no constraint on the amount of padding from a correctness standpoint. Nonetheless, it should be large enough to minimize missed checkpoints, yet not too large that the worst-case utilization is needlessly inflated.

We set the padding equal to the maximum WCET among all sub-tasks in the task. Using this heuristic ensures that a pipeline with exactly the same timing as the VISA does not miss any checkpoints. The example in Figure 4 demonstrates our point. For sake of argument, the processor being checked is the VISA itself. There are four sub-tasks, the third having the largest WCET. The task's WCET is padded by the third sub-task's WCET. While the first two and last checkpoints are quite lax, the third sub-task just barely meets its checkpoint.

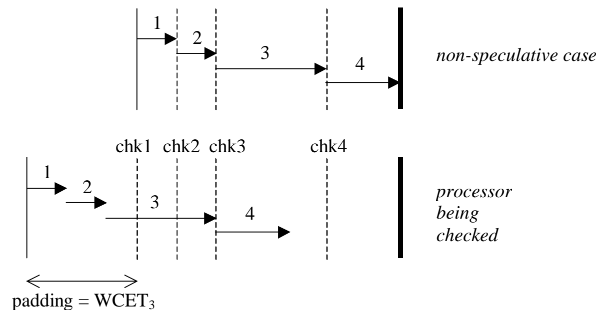

Figure 4. Rationale for padding WCET with the maximum WCET among all sub-tasks.

4.4 Handling preemptions

The current task may be preempted when another task is released. A release causes the task scheduler to be invoked, interrupting the current task. The task scheduler then makes a decision to either schedule a new task (preemption) or continue the current task. If the current task is preempted, the content of the watchdog counter is saved along with the current pipeline mode (complex or simple mode). When the preempted task is later resumed, its watchdog counter value is restored so that its progress can

continue to be monitored, if necessary. If the pipeline mode does not match the mode required by the resumed task, then the pipeline is reconfigured accordingly before resuming the task.

4.5 Budgeting scheduler overhead

The task scheduler is invoked when a task completes or a task is released. The scheduler itself must be accounted for in the worst-case utilization. This is true for both VISA and non-VISA systems.

Considering only periodic hard-real-time tasks, the number of scheduler invocations is bounded by two times the number of tasks, since it is invoked when a task is released or when a task completes. We account for the scheduler in the worst-case utilization indirectly, by padding the WCET of the first and last sub-tasks of each task with the scheduler's WCET. Padding the first sub-task of a task accounts for the scheduler invocation when the task is released. Padding the last sub-task of a task accounts for the scheduler invocation when the task completes. There is another important benefit of (logically) incorporating the task scheduler within tasks themselves. In particular, we can safely execute the scheduler on the complex processor, without dividing the scheduler into sub-tasks like we have to do with other tasks. Since the scheduler becomes part of another sub-task, the checkpoint of this other sub-task safely covers the sub-task plus scheduler.

Similar padding needs to be done for conventional real-time systems (non-VISA systems). Since there are no sub-tasks, the same effect is achieved by padding each task's WCET with two times the WCET of the scheduler.

Sporadic hard-real-time tasks also invoke the scheduler when they are released and completed. Earlier, we indicated that these tasks are not accounted for in the worst-case utilization *a priori*. Nonetheless, the overhead of invoking the scheduler when a sporadic task is released does need to be budgeted. Otherwise, it would be unsafe just to perform the acceptance test, regardless of the outcome of the test. The scheduler must also limit the number of sporadic tasks that will be considered for acceptance over a given time interval. Once the number of acceptance tests reaches the limit, no more sporadic tasks will even be considered until the next interval. Once again, these issues are universal to all real-time systems, VISA and non-VISA alike. Thus, in this paper, the same methodology is applied to both.

4.6 SMT considerations

The complex processor may have SMT capability. On the other hand, the VISA specification does not include SMT. In this section, we describe what must be done to disable SMT when the pipeline is reconfigured to operate in the simple mode. There are two steps to disable SMT. First, in-flight instructions not belonging to the periodic or sporadic hard-real-time task are drained or squashed from the pipeline. Second, other tasks are suspended – not swapped out – by disabling fetching of new instructions from these tasks. The same method is used when a preempted task is resumed, and SMT is currently active but the resumed task requires the simple mode of operation.

5 Experimental Methodology

5.1 VISA framework

The VISA framework consists of (1) the VISA specification, (2) the VISA-compliant complex processor, and (3) the static worst-case timing analysis tool for deriving WCETs based on the VISA. We use the same framework as described in the literature [1].

As shown in Table 1, the VISA specification describes an in-order scalar pipeline with static branch prediction and split L1 instruction and data caches.

Table 1. VISA specification and VISA-compliant complex processor.

	VISA specification	**Complex Pipeline**
Pipeline Stages	fetch, decode, register read, execute, memory, writeback	fetch, dispatch, issue, reg. read, execute or agen/memory, writeback, retire
Fetch Unit	1 instr./cycle	4 instr./cycle
	static branch prediction: BT/FNT heuristic	dynamic branch prediction: 64K-entry *gshare*, 16-bit BHR
Execution Core	in-order execution scalar execution (1 instr./cycle) 1 unpipelined univ. func. unit	out-of-order execution 4-way superscalar (4 instr./cycle) 4 pipelined univ. function units 128-entry reorder buffer 64-entry issue queue 64-entry load/store queue 2 ports to load/store queue + D$
Memory Hierarchy	L1 I-cache: 64KB, 4-way set-assoc., 64B block, 1 cycle hit L1 D-cache: 64KB, 4-way set-assoc., 64B block, 1 cycle hit	
	100 cycles worst-case memory stall time	100 cycles minimum stall time

Microarchitecture parameters of the VISA-compliant complex processor are also shown in Table 1. It is a 4-issue dynamically-scheduled superscalar processor with a *gshare* branch predictor [17] and the same L1 cache configuration as the VISA. The complex processor can be reconfigured to a simple mode of operation that directly implements the VISA. Pipeline stage modifications for accommodating the simple mode are described in depth in the literature [1].

We use a state-of-the-art static worst-case timing analysis tool to automatically derive the WCETs of tasks on the VISA. WCETs will be presented in Section 5.4. Details regarding the tool can be found in the literature [1].

5.2 Cycle-accurate simulator and processor models

A multi-tasking environment is modeled using a cycle-accurate simulator built using the SimpleScalar toolset [3]. The simulator supports three different processor models. *Complex* is the VISA-compliant complex processor, featuring the complex pipeline that can be dynamically reconfigured to the simple mode of operation. *Complex-smt* is like *complex*, enhanced with SMT capability (up to 8 threads). *Simple-fixed* is the explicitly-safe simple processor, i.e., a literal (fixed) implementation of the VISA.

5.3 Multi-tasking support

We model the software and hardware of the multi-tasking system in significant detail. Each task has its own binary, including the scheduler itself. Thus, scheduler overhead is included by virtue of executing it like any other task. The scheduler is invoked when a task completes or when a task is released. Task releases are implemented by the scheduler using a count-down timer provided by the processor, called the release timer. Before turning the processor over to a task, the scheduler sets the release timer to interrupt the processor at the next task release.

5.4 Benchmarks and tasksets

We use six different benchmarks from the C-lab real-time benchmark suite [5], shown in Table 2. The C-lab benchmarks are used extensively in WCET research, in particular because irregular program features that complicate static timing analysis are explicitly avoided, which is common in hard real-time code. The benchmarks are compiled with –O3 optimization enabled.

Table 2. Benchmark description.

benchmark	# sub-tasks	WCET (ms)	Padded WCET (ms)	Ave. exec. time (ms)	
				simple-fixed	*complex*
adpcm	16	3.35	3.71	2.43	0.64
cnt	10	0.16	0.18	0.07	0.02
fft	10	0.59	0.66	0.36	0.06
lms	20	0.19	0.2	0.17	0.04
mm	20	2.25	2.36	2.10	0.66
srt	20	3.53	3.71	1.82	0.55

The first column in Table 2 shows the number of sub-tasks in each task. Sub-task selection was done manually. The second column shows the WCET produced

by the static timing analyzer. Since *simple-fixed* is non-speculative in terms of safety, it presents this WCET unmodified to the scheduler. *Complex* is speculative in terms of safety, so it presents a padded WCET to the scheduler (as described in Section 4.3), shown in the third column. Note that all WCETs (for both *complex* and *simple-fixed*) are padded with the scheduler overhead as explained in Section 4.5. The last two columns show the average execution times on *simple-fixed* and *complex*. The speedup of *complex* with respect to *simple-fixed* is between 3.2 (mm) and 6.0 (fft) with all but one task (fft) in the 3 to 4 speedup range.

We created base tasksets by combining pairs of periodic hard-real-time tasks, shown in Table 3. In the results section, these base tasksets are augmented with sporadic hard-real-time, periodic soft-real-time, and non-real-time tasks. Only the base tasksets are shown in the table because only periodic hard-real-time tasks affect the worst-case utilization. Tasksets are named by enumerating the first letters of hard real-time tasks, the two periodic tasks listed first followed by sporadic tasks if present. Example: ASL uses *adpcm* and *srt* as the two periodic hard-real-time tasks and *lms* as a sporadic hard-real-time task. MPEG and *gcc* (the latter from SPEC2K) are always used for periodic soft-real-time and non-real-time tasks, respectively, and so are not enumerated explicitly. Periods of the periodic hard-real-time tasks (P_1 and P_2) were chosen to achieve a worst-case utilization (U_{max}) of 1 for *complex*. Notice that the worst-case utilization for *simple-fixed* is less, because its WCETs are not padded. Thus, *simple-fixed* initially has an advantage by virtue of having static slack in the schedule.

Table 3. Base tasksets.

base	$task_1$		$task_2$		U_{max}	
taskset	name	P_1(ms)	name	P_2(ms)	*simple-fixed*	*complex*
AS	adpcm	4.64	srt	19.39	0.90	1.0
AM	adpcm	19.37	mm	2.95	0.93	1.0
MS	mm	2.95	srt	19.39	0.94	1.0
FC	fft	0.83	cnt	0.91	0.89	1.0
MC	mm	2.95	cnt	0.91	0.94	1.0
AL	adpcm	4.64	lms	1.02	0.90	1.0
LF	lms	0.25	fft	3.45	0.93	1.0
ML	mm	2.95	lms	1.02	0.94	1.0
LC	lms	0.25	cnt	0.91	0.93	1.0
SL	srt	4.41	lms	1.02	0.94	1.0

6 Experimental Results

We simulated all base tasksets (composed of only periodic hard-real-time tasks) for 100 ms. The first key result is that no hard deadlines are missed, for both *simple-fixed* and *complex*, as expected. In the sections that follow, we present results of adding sporadic, soft-real-time, and non-real-time tasks to the base tasksets, using a fixed simulation time of 100 ms. In all cases, no hard deadlines were missed. In

some of the SMT runs, *complex-smt* misses checkpoints and, in spite of these, the VISA protection mechanism successfully enforces hard deadlines.

6.1 Sporadic tasks

In this section, we add a single sporadic hard-real-time task to each of the base tasksets. The last letter enumerated in the taskset name corresponds to the sporadic task. The sporadic task is randomly released. As described in Section 4.5, we limit the number of acceptance tests to 100 in the 100 ms simulation timeframe, so that the acceptance test can be accounted for in the utilization. (Utilization by the acceptance test is already included in the worst-case utilizations reported in Section 5.4.)

Since *complex* finishes periodic hard real-time tasks much faster than *simple-fixed* (generating considerable slack in the schedule), the likelihood of a sporadic task getting accepted is greater for *complex* than *simple-fixed*. The results confirm this. Figure 5a shows the percentage of sporadic tasks accepted by *simple-fixed* and *complex* for various tasksets. As expected, *complex* accepts 2.4 to 16 times more sporadic tasks than *simple-fixed*. Figure 5b shows that the residual idle time remains high even with sporadic tasks, for *simple-fixed* (due to low acceptance) and *complex* (due to low actual utilization by them). However, *complex* exhibits far more residual idle time, which can be used for soft real-time tasks.

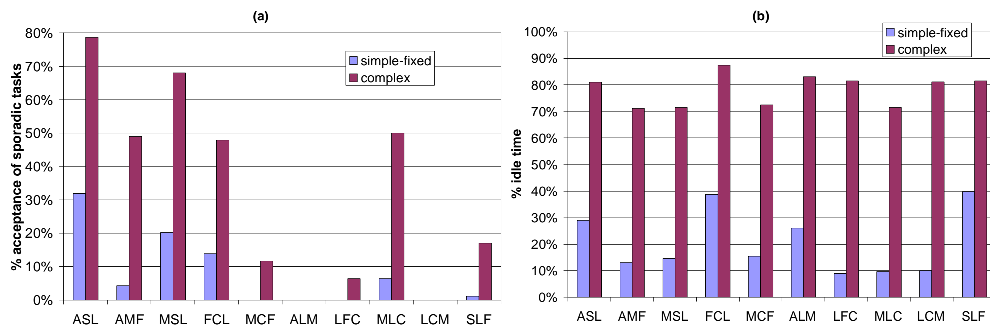

Figure 5. (a) Percentage of sporadic tasks accepted and (b) idle time percentage.

An interesting observation is that tasksets with large periodic tasks yield higher acceptance rates than tasksets with small periodic tasks, for the same sporadic task and same worst-case utilization. For a given utilization, a small periodic task has a shorter period compared to a large periodic task, i.e., it is released more frequently. Thus, a single release of the large periodic task corresponds to multiple evenly distributed releases of the small periodic task. The large task completes quickly on the complex processor,

making a large amount of idle time evident. On the other hand, while the first instance of the small task finishes even quicker, there are many remaining releases and hence future WCETs. These remaining releases effectively defer the accrual of slack, since the complex processor is not allowed to dispense with the WCETs until the releases. This effect is evident in our results. For example, although ASL (adpcm and srt are large) and FCL (fft and cnt are small) have the same sporadic task, ASL accepts more sporadic tasks (79%) compared to FCL (48%).

Figure 5a also shows that tasksets with small sporadic tasks (lms, cnt, and fft) have a higher acceptance rate than tasksets with large sporadic tasks (mm, srt, and adpcm). This is to be expected, since smaller sporadic tasks are more likely to fit in the schedule at any given moment. We further investigate the impact of the size of sporadic tasks (in terms of WCET) on the acceptance rate. For a given base taskset, we attempt three successively larger sporadic tasks. Note that the worst-case utilization is not affected by the size of the sporadic task since it is not considered *a priori*. Figure 6 shows the percentage of sporadic tasks accepted by *simple-fixed* and *complex* for successively larger sporadic tasks in three base tasksets (FC, AS, AL). For all base tasksets, the acceptance rate decreases for both *simple-fixed* and *complex* as the sporadic task gets larger. Since the slack available to schedule the sporadic task is the same in all three cases, the acceptance rate is higher for small sporadic tasks compared to large ones. Nonetheless, *complex* still has a higher acceptance rate than *simple-fixed*, even for large sporadic tasks.

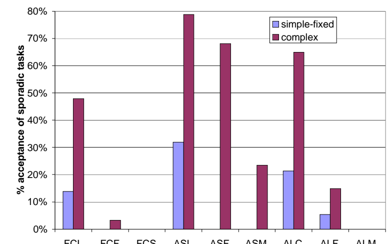

Figure 6. Acceptance rate for successively larger sporadic tasks, for 3 base tasksets (FC, AS, AL).

6.2 Soft real-time tasks

We now consider a mixed system with two periodic hard-real-time tasks and one periodic soft-real-time task. Sporadic tasks are not included in these experiments. The periodic soft-real-time task is an MPEG decoder. MPEG frames are released at the constant rate of 300 frames/second, which yields 30 total frames in our 100 ms simulation timeframe. If a new frame is released before the current frame is done, the new frame is dropped.

Figure 7a shows the percentage of MPEG frames successfully serviced by *simple-fixed*, *complex,* and *complex-smt*. For *complex-smt*, we investigate three different fetch policies: priority for the hard real-time task (PRIO), round-robin (RR), and ICOUNT (ICNT) [24]. For the PRIO policy, when choosing among multiple soft real-time tasks (we consider more than one soft real-time task later), we give priority to the one with the earliest deadline.

Complex and all *complex-smt* models always perform better than *simple-fixed. Simple-fixed* drops 63% to 90% of MPEG frames. No MPEG frames are dropped by *complex, complex-smt (RR)*, and *complex-smt (PRIO)* for all 10 tasksets. However, interestingly, 3% to 27% of frames are dropped by *complex-smt (ICNT).* The ICOUNT fetch policy causes some checkpoints to be missed by the currently-executing hard real-time task. Apparently, in several cases, the soft real-time task steals enough resources away from the hard real-time task to cause it to be less timely than the VISA. The processor reverts to the simple mode of operation to ensure the WCET bound is still respected. This significantly delays completion of the current MPEG frame (the thread is suspended) and the next released frame gets dropped. Note that although the hard real-time task missed some checkpoints, no hard deadlines were compromised as should be the case with VISA protection mechanisms in place. We conclude that policies that attempt to maximize throughput (such as ICOUNT) should be balanced with the need for minimum forward progress by the hard real-time task – just enough to be as timely as the VISA.

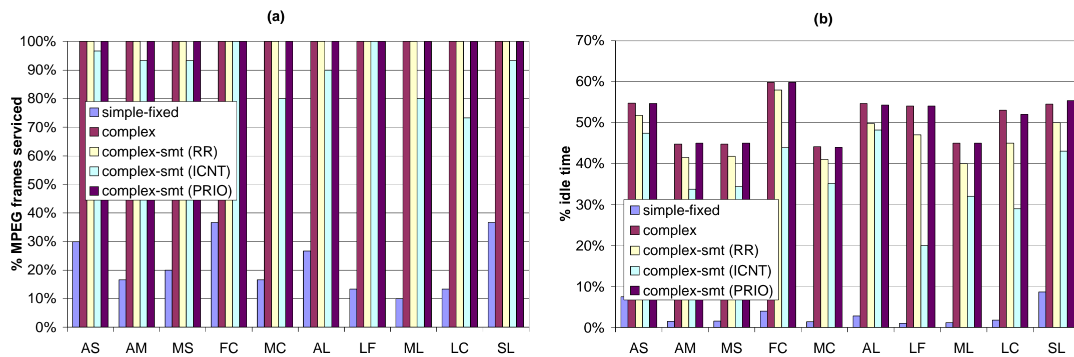

Figure 7. (a) Percentage of MPEG frames successfully serviced and (b) idle time percentage. (Results are for a single periodic soft-real-time task.)

Figure 7b shows the residual idle time percentage for the various models. *Complex* and *complex-smt (PRIO)* have the most residual idle time (51% and 51% on average). *Complex-smt (RR)* and *complex-smt (ICNT)* are next, with average idle time percentages of 47% and 37%, respectively.

Next, we consider two periodic soft-real-time tasks (two MPEG tasks) instead of only one. Figure 8a shows the percentage of MPEG frames successfully serviced for the two MPEG tasks, with frame rates of 25 and 30 frames per 100ms, respectively. Though the complex models service more frames than *simple-fixed,* 6% to 89% frames are dropped by the complex models. This is due to contention between the two MPEG tasks as well as missed checkpoints. As before, *complex* and *complex-smt (PRIO)* have the highest percentage of MPEG frames serviced (91% and 89%, on average), followed by *complex-smt (RR)* (69% on average), and *complex-smt (ICNT)* (20% on average).

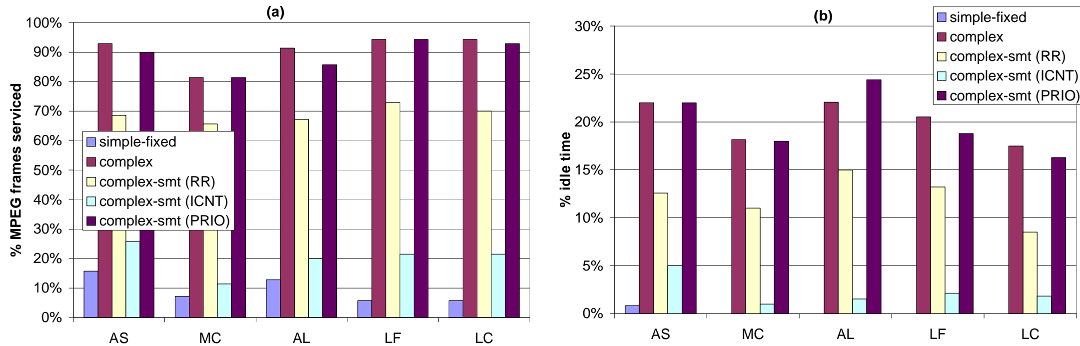

Figure 8. (a) Percentage of MPEG frames successfully serviced, and (b) idle time percentage. (Results are for two periodic soft-real-time tasks.)

6.3 Sporadic and soft real-time tasks

We now consider mixed systems with two periodic hard-real-time tasks, one sporadic hard-real-time task, and one periodic soft-real-time task.

Figure 9a shows the percentage of sporadic tasks accepted by *simple-fixed, complex,* and *complex-smt* (for three fetch policies). All complex models have higher acceptance rates than *simple-fixed*, for all tasksets. *Complex-smt (RR)* and *complex-smt (ICNT)* have lower acceptance rates compared to *complex* and *complex-smt (PRIO).* SMT slows the execution of individual tasks, including the periodic hard-real-time tasks. The acceptance test estimates the amount of worst-case work remaining among released hard real-time tasks, taking into consideration already-completed sub-tasks. *Complex-smt (RR)* and *complex-smt (ICNT)* have lower acceptance rates because they exhibit more residual worst-case work as compared to *complex* and *complex-smt (PRIO).* By slowing execution of periodic hard-real-time tasks, *complex-smt (RR)* and *complex-smt (ICNT)* indirectly prioritize soft real-time tasks over sporadic tasks.

Figure 9b shows percentages of MPEG frames serviced. *Simple-fixed* has the lowest percentage of serviced frames. No frames are dropped by *complex, complex-smt (RR),* and *complex-smt (PRIO)*, whereas 3% to 27% of frames are dropped by *complex-smt (ICNT).* As observed in the previous section, *complex-smt (ICNT)* induces from 1 to 30 missed checkpoints, causing the complex processor to switch to simple mode as often. This causes the current frame to be deferred and the next frame to be dropped. Indeed, further analysis revealed that dropped frames were those released just after mispredictions.

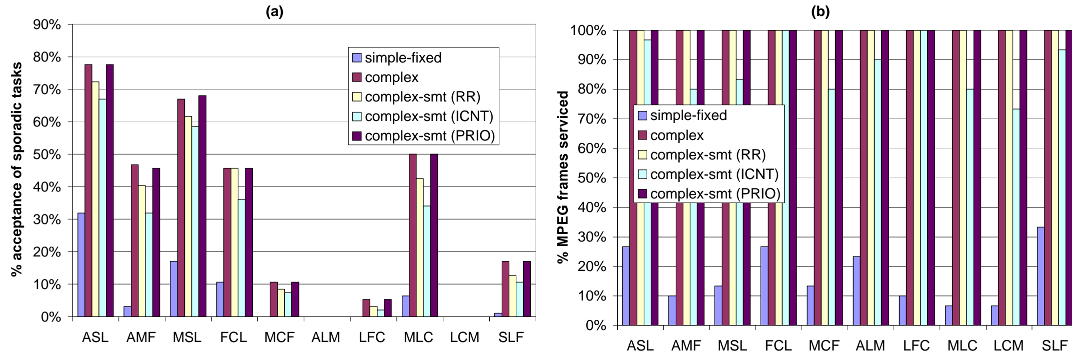

Figure 9. (a) Percentage of sporadic tasks accepted and (b) percentage of MPEG frames serviced.

6.4 Sporadic, soft real-time, and non-real-time tasks

There is still considerable unreclaimed slack in the VISA-compliant complex models. Accordingly, in this section, we use the same mix of real-time tasks as in the previous section but add a non-real-time "background" application. This gives tasksets composed of two periodic hard-real-time tasks, one sporadic hard-real-time task, one periodic soft-real-time task, and a background application. For the background application, we use the *gcc* benchmark from the SPEC2000 integer benchmark suite.

Figure 10a-c shows sporadic acceptance rate, percentage of MPEG frames successfully completed, and number of non-real-time (*gcc*) instructions committed, respectively. For sporadic tasks and MPEG frames, we observe the same trends as before. As expected, the complex models commit from 31 to 201 times more instructions of the background application than *simple-fixed. Complex-smt (RR)* and *complex-smt (ICNT)* drop more MPEG frames compared to *complex* and *complex-smt (PRIO).* This is because the soft real-time task is given priority over the background application in *complex* and *complex-smt (PRIO).* This is also reflected in the number of committed instructions of the background application.

On average, *complex-smt (RR)* and *complex-smt (ICNT)* commit 6 times and 4 times more *gcc* instructions than *complex*, and 4 times and 2 times more than *complex-smt (PRIO)*.

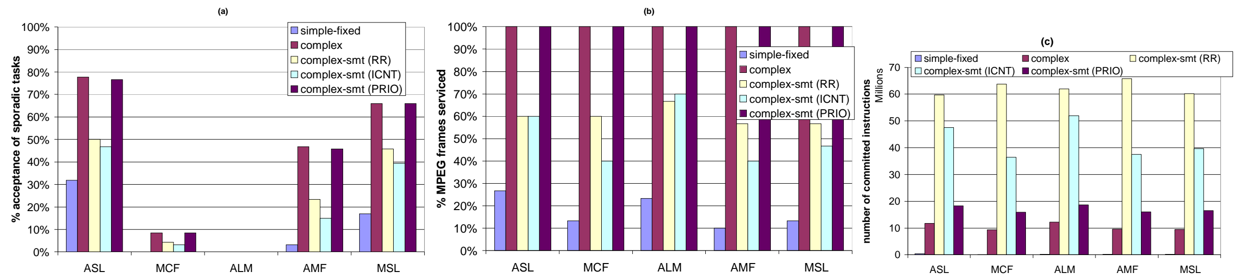

Figure 10. (a) Percentage of sporadic tasks accepted, (b) percentage of MPEG frames successfully serviced, and (c) number of instructions committed for the non-real-time task.

7 Summary and Future Work

Bounding worst-case execution times (WCET) of tasks on contemporary processors is difficult, and beyond the capabilities of current static timing analysis tools. While we expect these tools to improve, it is difficult to keep up with microprocessor enhancements. The VISA framework expedites the introduction of unsafe processors in safe (hard) real-time systems, by providing a simple architectural model to static timing analysis and dynamically confirming that the derived WCET bounds are adhered to. Interim checkpoints facilitate verification of the unsafe processor's progress. As a fall-back mode, if a checkpoint is missed, the unsafe pipeline features a coarse-grain reconfiguration technique whereby complex features are disabled to directly implement the VISA.

In this paper, we showed that the high-performance processor does not reduce worst-case utilization with respect to an explicitly-safe simple processor since WCETs are based on the VISA, but that significant benefits can be derived from the reduction in actual utilization. The contribution of this paper is exploring mixed-task systems (hard/soft, periodic/sporadic), which are able to exploit the throughput advantages of the VISA-compliant complex processor (unlike systems with only periodic hard-real-time tasks). In a mixed system, 2.4-16 times more sporadic hard-real-time tasks are accepted compared to an explicitly-safe processor, and no soft-real-time tasks are dropped whereas the explicitly-safe processor drops 67-93% of soft-real-time tasks.

With SMT and a round-robin fetch policy, soft-real-time tasks slow the execution of periodic hard-real-time tasks, making it more difficult to accept sporadic hard-real-time tasks. The same is true for

the ICOUNT fetch policy. Moreover, the narrow focus on maximizing throughput causes checkpoints to be missed on occasion. This invokes the simple mode of operation, deferring the current soft-real-time task and thereby causing the next soft-real-time task to be dropped. These issues are resolved when the SMT fetch policy prioritizes the periodic hard-real-time task, making the sporadic acceptance test more likely to succeed and eliminating missed checkpoints.

For future work, we would like to find ways to leverage the VISA concept for scheduling more periodic hard-real-time tasks. This will require means to overlap WCETs such that the resulting execution exceeds the equivalent of a fully-utilized single-threaded uniprocessor under WCET assumptions. SMT principles may provide a starting point. We would also like to integrate the soft real-time scheduling techniques for SMT processors developed by Jain *et al.* [10] in a VISA framework.

8 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 0310860. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

9 References

[1] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (visa): Exceeding the complexity limit in safe real-time systems. 30th Int'l Symp. on Computer Architecture, June 2003.

[2] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. IEEE Real-Time Systems Symposium, pp. 172-181, Dec. 1994.

[3] D. C. Burger, T. M. Austin, and S. Bennett. The Simplescalar Tool Set, Version 2.0. Tech. Rep. 1342, Computer Science Department, University of Wisconsin-Madison, 1997.

[4] Giorgio C. Buttazzo. Hard Real-Time Computing Systems. Kluwer, 1997.

[5] C-Lab WCET Benchmarks. Available from http://www.c-lab.de/home/en/download.html.

[6] T. Hand. Real-time systems need predictability. Computer Design (RISC Supplement), pp. 57-59, Aug. 1989.

[7] M. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. IEEE Real-Time Systems Symposium, pp. 68-77, Dec. 1992.

[8] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. IEEE Real-Time Systems Symposium, pp. 288-297, Dec. 1995.

[9] Y. Hur et al. Worst case timing analysis of RISC processors: R3000/R3010 case study. Real-Time Sys. Symp., Dec. 1995.

[10] R. Jain, C. J. Hughes, and S. V. Adve. Soft real-time scheduling on simultaneous multithreaded processors. 23rd Int'l Real-Time Systems Symposium, Dec. 2002.

[11] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. Real-Time Tech. & App. Symp., Jun 1996.

[12] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. IEEE Real-Time Systems Symposium, pages 298--397, Dec. 1995.

[13] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. IEEE Real-Time Systems Symposium, pages 254--263, Dec. 1996.

[14] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. IEEE Real-Time Systems Symposium, pages 97-108, Dec. 1994.

[15] J. Liu. Real-Time Systems. Prentice Hall, 2000.

[16] T. Lundqvist and P. Stenstrom. An integrated path and timing analysis method based on cycle-level symbolic execution. Real-Time Systems, 17(2/3):183-208, Nov. 1999.

[17] S. McFarling. Combining Branch Predictors. Tech. Rep. TN-36, WRL, June 1993.

[18] F. Mueller. Timing analysis for instruction caches. Real-Time Systems, 18(2/3):209-239, May 2000.

[19] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. Real-Time Systems, 5(1):31-61, March 1993.

[20] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. Real-Time Systems, 1(2):159-176, Sep. 1989.

[21] K. Seth, A. Anantaraman, E. Rotenberg, and F. Mueller. Safe real-time scheduling on contemporary processors exploiting a virtual simple architecture (visa). *Submitted to RTSS'03*.

[22] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. Deadline Scheduling for Real-Time Systems. Kluwer, 1998.

[23] H. Theiling and C. Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. IEEE Real-Time Systems Symposium, pp. 144-153, Dec. 1998.

[24] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. 23rd Int'l Symp. on Comp. Arch., May 1996.

[25] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. Real-Time Systems, 5(4):319-343, Oct. 1993.