Enforcing Safety of Real-Time Schedules on Contemporary Processors Using a Virtual Simple Architecture (VISA)

Aravindh Anantaraman†, Kiran Seth‡, Eric Rotenberg† and Frank Mueller†
† Departments of Computer Science/Electrical and Computer Engineering
Center for Embedded Systems Research
North Carolina State University, Raleigh, NC 27695
‡ Qualcomm, Inc., 2000 Center Green Way, Cary, NC 27513
avananta@ece.ncsu.edu, krseth@qualcomm.com, ericro@ece.ncsu.edu, mueller@cs.ncsu.edu

ABSTRACT

Determining safe and tight upper bounds on the worst-case execution time (WCET) of hard real-time tasks running on contemporary microarchitectures is a difficult problem. Current trends in microarchitecture design have created a complexity wall: By enhancing performance through ever more complex architectural features, systems have become increasingly hard to analyze.

This paper extends a framework, introduced previously as Virtual Simple Architecture (VISA), to multi-tasking real-time systems. The objective of VISA is to obviate the need to statically analyze complex processors by instead shifting the burden of guaranteeing deadlines – in part – onto the hardware. The VISA framework exploits a complex processor that ordinarily operates with all of its advanced features enabled, called the complex mode, but which can also be downgraded to a simple mode by gating off the advanced features. A WCET bound is statically derived for a task assuming the simple mode. However, this abstraction is speculatively undermined at run-time by executing the task in the complex mode. The task's progress is continuously gauged to detect anomalous cases in which the complex mode underperforms, in which case the processor switches to the simple mode to explicitly enforce the overall contractual WCET. The processor typically operates in complex mode, generating significant slack, and the VISA safety mechanism ensures bounded timing in atypical cases. Extra slack can be exploited for reducing power consumption and/or enhancing functionality.

By extending VISA from single-task to multi-tasking systems, this paper reveals the full extent of VISA's powerful abstraction capability. Key missing pieces are filled in: (1) preserving integrity of the gauging mechanism despite disruptions caused by preemptions; (2) demonstrating compatibility with arbitrary scheduling and dynamic voltage scaling (DVS) policies; (3) formally describing VISA speculation overheads in terms of padding tasks' WCETs; and (4) developing a systematic method for minimizing these overheads. We also propose a novel VISA variant that dynamically accrues the slack needed to facilitate speculation in the complex mode, eliminating the need to statically pad WCETs and thereby enabling VISA-style speculation even in highly-utilized systems.

1. INTRODUCTION

Determining safe (never underestimated) but tight upper bounds on the worst-case execution time (WCET) of hard real-time tasks running on contemporary microarchitectures is a difficult problem. Schedulability tests based on real-time theory depend on safe upper bounds on the WCET to guarantee that deadlines are met [20, 7, 30]. Safe and tight WCET bounds can be provided by static timing analysis for relatively simple, scalar architectures with in-order execution, static branch prediction (if any branch prediction at all), and split caches as well as locking caches [23, 19, 31, 11, 21, 32].

Current trends in microarchitecture design have created a complexity wall: By enhancing performance through ever more complex architectural features, systems are increasingly hard to analyze. While static timing analysis techniques have been developed to safely and tightly bound the WCET for simple architectures, technology is advancing faster than timing analysis techniques.

Our work approaches timing analysis from a hardware/software co-design angle. We build on a previously introduced framework named Virtual Simple Architecture (VISA) [1, 28]. The VISA framework provides a simple processor model to timing analysis (e.g., a simple, in-order, scalar pipeline with static branch prediction and split instruction/data caches). This simple model is only an *abstraction* that allows us to derive safe and tight WCET bounds with existing static timing analysis tools. In actuality, the underlying processor is arbitrarily complex (e.g., a complex, out-of-order, superscalar pipeline with dynamic branch prediction and split instruction/data caches). To ensure the simple timing abstraction is honored, a task's progress on the complex processor is continuously gauged via intermediate timing checkpoints, which, if not exceeded, reflect timely on-going progress. A missed checkpoint is safely recoverable by reconfiguring the complex processor into a downgraded simple mode of execution to explicitly bound remaining execution time, by disabling unsafe microarchitectural features (e.g., overriding dynamic prediction with static prediction, disabling all but one superscalar way for scalar execution, blocking instructions from issuing out-of-order, etc.). Thus, a single pipeline can be configured to operate in either an unconstrained *complex mode* or a downgraded *simple mode* that matches the VISA specification, the latter only used in anomalous cases where complex execution may exceed its simple counterpart. For example, the complex mode may (temporarily) underperform its simpler counterpart as a result of extra branch mispredictions caused by excessively long branch predictor training times, branch predictor table aliasing, etc., coupled with costlier misprediction penalties for backing out of deep speculation.

The added cost of supporting the simple mode is minor since it does not represent any additional functionality, only the gating-off of existing high-performance features. Moreover, operating primarily in the complex mode is significantly more energy efficient than the alternative (using an explicitly-safe simple processor), since the same performance can be achieved at a far lower frequency/voltage by compensating with abundant instruction-level parallelism (ILP).

Our prior work [1] implemented the VISA framework in systems with only one periodic task. We demonstrated two-fold benefits.

Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004)
1052-8725/04 $20.00 © 2004 IEEE

(1) The VISA framework enables the safe use of contemporary processors in hard real-time systems by enforcing an agreed-upon WCET abstraction of a single task. (2) The tangible benefit is the creation of large amounts of dynamic slack due to the high-performing complex mode. The spare capacity can be exploited either to increase functionality (i.e., schedule other soft-real-time and non-real-time tasks) [2], or save power/energy via dynamic voltage scaling (DVS) [1].

This prior VISA research considers only single-task systems, making it incomplete in many aspects and also veiling the full extent of VISA's powerful abstraction capability. Key gaps exist, including: enforcement of tasks' WCETs in the context of preemptive multi-tasking; compatibility with arbitrary scheduling and DVS policies and any other real-time software components; explicit treatment of VISA speculation overhead including systematic overhead assessment. Moreover, we would like to explore new, efficient VISA variants for eliminating overheads that otherwise become limiters in highly-utilized systems. Accordingly, the major contributions of this paper are as follows.

- *Extension to multi-tasking*: As mentioned previously, the crux of VISA is preserving a contractual WCET abstraction of a single task. In this paper, we adapt the VISA framework for a multi-tasking system, which has a set of periodic tasks (i.e., a task-set). In a multi-tasking system, tasks can be interrupted (preempted) by higher-priority tasks and later resumed. Task preemptions disrupt the VISA gauging mechanism, potentially undermining enforcement of the WCET abstraction. We show that the VISA framework can be easily adapted to account for task preemptions by saving and restoring the state of the gauging mechanism (mode bit and watchdog counter) at task interruptions and resumptions, respectively. As such, this paper shows that the VISA framework correctly enforces the WCET abstraction in multi-tasking systems as well.
- *Transparency to real-time OS*: Multiple tasks are managed by the real-time OS, which includes task schedulers, DVS schedulers, etc. We must consider the implications of the VISA framework – if any – on these software components. This paper makes the case that the VISA framework is transparent by way of preserving the WCET abstraction that is the basis of all scheduling theory. We demonstrate the transparency principle by deploying a conventional EDF scheduler and an arbitrary DVS scheduler – the Look-Ahead DVS real-time scheduler proposed by Pillai and Shin [24] – on top of the VISA framework. Most importantly, we show that no VISA-specific or other modifications were needed to these software components.
- *Energy evaluation in multi-tasking systems*: New experiments show that the significant energy savings previously observed in single-task systems transfer to multi-tasking systems. Using the Look-Ahead DVS real-time scheduling scheme [24], we show that a VISA-protected complex processor consumes 19-58% less energy than an explicitly-safe simple processor.
- *Systematic and intuitive method for assessing and minimizing VISA speculation overhead*: At the start of a task, the VISA gauging mechanism requires some extra time to attempt the task in complex mode. This extra time is conceptually a "headstart" for the complex mode. There are three contributions here with respect to our previous VISA work. In our prior work, the notion of the headstart amount was implied but not prominent. The first contribution is explicitly demonstrating the need for a headstart and ensuring overall system safety by padding each task's headstart amount to its WCET. The headstart amount is arbitrary from a correctness standpoint, i.e., the choice of headstart does not affect safety of the VISA framework. Nonetheless, the headstart should be chosen carefully: a headstart that is too large over-inflates the WCET, whereas a headstart that is too small increases the likelihood of missing checkpoints even under anomaly-free behavior. The second contribution is that we provide a systematic and intuitive approach to assess a minimal headstart amount. Third, we show that the headstart amount tends to be small compared to the overall WCET, thus having little effect on overall worst-case utilization.
- *Novel zero-overhead VISA speculation approach – dynamic headstart accrual*: Although we demonstrate that the headstart overhead tends to be small, task-sets with near-100% worst-case utilization may not be able to accommodate the static headstart padding, precluding speculation in the complex mode altogether, thus defaulting to only the simple mode. To address this, we propose a novel approach called *dynamic headstart accrual* (as opposed to the explicit padding approach). In this approach, a task is started in the simple mode. Since actual execution time is typically less than the WCET, dynamic slack accrues even while operating in the simple mode. Once the accumulated slack exceeds the required headstart amount, speculation becomes viable once again. At such time, the pipeline is reconfigured to operate in complex mode, the VISA protection mechanisms are engaged, and the task is speculatively executed in complex mode.

In section 2, we review the VISA gauging mechanism, overall VISA system design, and benefits of the VISA framework. In section 3, we explain VISA speculation overhead and develop a systematic method of assessing the headstart amount. We present the novel, zero-overhead VISA speculation approach, called dynamic headstart accrual, in Section 4. Section 5 describes how arbitrary DVS algorithms can be transparently employed in the VISA framework. Section 6 describes modifications to the VISA framework to safely accommodate preemptions. The simulation environment and benchmarks are described in Section 7. Results and analyses are presented in Section 8. Finally, related work is discussed in Section 9, and Section 10 summarizes the paper.

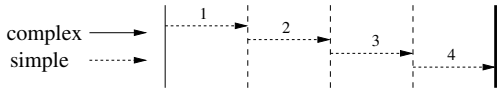
2. VISA FRAMEWORK

Let us first review the main concepts of the VISA framework. In VISA, a WCET bound is derived for a task assuming the simple execution mode. This WCET bound should be derived by some means that guarantees safety, such as a static timing analysis tool [11] or pWCET for probabilistic approaches [4]. While the WCET is determined for the simple mode, the task is executed using the complex mode at run-time. The complex mode must be considered unsafe since, strictly speaking, the WCET bound does not apply to it. In practice, the complex mode is much faster on the average, but we need to confirm this dynamically since there may exist certain instances where complex execution exceeds its simple counterpart. This is achieved by dividing the task into multiple smaller sub-tasks in software and gauging their progress in hardware. Sub-tasks are assigned soft deadlines, called checkpoints. Continued safe progress in complex mode is confirmed as long as sub-tasks meet their checkpoints. If a checkpoint is missed, the processor is

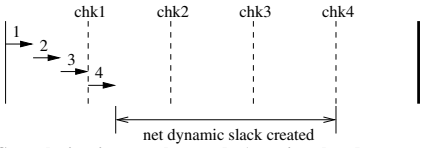
reconfigured to operate in simple mode, thereby bounding execution time explicitly.

2.1 Gauging Progress via Checkpoints

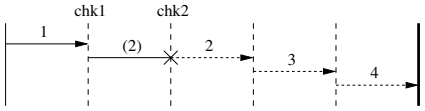
Our method is illustrated in Figures 1(a)-1(c) for a task with four sub-tasks, where actual execution times are indicated by arrows. Fig. 1(a) shows the conventional, conservative approach in which complexity is disabled in order to be explicitly safe. The task is executed entirely in simple mode. Dashed vertical lines in this figure indicate the WCETs of sub-tasks, which are very tight in the example.



(a) Non−speculative simple mode.



(b) Speculation in complex mode (requires headstart padding).



(c) Speculation in complex mode with mode switch at misprediction in sub−task 2.

Fig. 1: Gauging progress under different scenarios.

Fig. 1(b) shows our *speculative approach* where the task is executed in the unsafe, complex mode. Sub-task checkpoints are indicated (chk1-chk4). Notice that the checkpoint for sub-task 1 (chk1) lines up with the *start time* of sub-task 1 in the non-speculative case (Fig. 1(a)); other sub-task checkpoints are similar. This leaves enough time to finish sub-task 1 plus execute sub-tasks 2, 3, and 4 in the simple mode if sub-task 1 misses its checkpoint in the complex mode. Thus, extra time must be budgeted relative to the non-speculative case, providing a "headstart" that facilitates gauging progress in the complex mode. In our initial approach [1], a task's WCET is explicitly padded to accommodate the headstart. Thus, the price for speculation is a higher theoretical worst-case utilization — but only slightly, as we will show in the results section. Fortunately, overall, speculation pays off substantially. As shown in Fig. 1(b), sub-tasks ordinarily complete well in advance of their checkpoints, creating substantial dynamic slack in the schedule. Dynamic slack can be exploited to reduce power/energy via DVS [1] or enhance functionality via scheduling of non-real-time or soft-real-time tasks [2].

While sub-tasks ordinarily complete well in advance of their checkpoints, we cannot statically prove it. Fig. 1(c) shows a counterexample, which may occur due to anomalous scenarios described in the introduction, for instance. Fig. 1(c) shows a sub-task *misprediction* for sub-task 2, indicated by an X at its checkpoint (chk2). In missing the checkpoint, its execution time using the complex mode can no longer be bounded. Hence, the processor switches to simple mode to explicitly bound the execution time of the unfinished sub-task and remaining sub-tasks (dashed arrows indicate simple mode). Here, we conservatively bound the residual execution time of the unfinished sub-task using its WCET since we cannot know how much of the sub-task was completed in the complex mode. The implication is that, as touched upon previously, the checkpoint for a sub-task is based on the cumulative WCETs of all later sub-tasks *plus the WCET of the sub-task itself*, ensuring enough time to safely complete remaining work in simple mode.

Notice that the VISA speculation overhead is incurred whether or not a misprediction occurs, as shown in Figures 1(b)—1(c).

2.1.1 Partitioning Tasks and Setting Checkpoints

Timeliness of the complex mode is gauged by dividing a task into smaller sub-tasks. Sub-tasks are currently identified by manually unrolling the outermost loop of the respective benchmarks. Alternatively, sub-task selection could be compiler-assisted in conjunction with a WCET tool to assess the preferred level of granularity of sub-tasks.

Each sub-task is assigned a soft deadline, a so-called checkpoint. Should a checkpoint be missed, an indication that the complex mode resulted in unsafe timing, execution will only resume after the architecture is reconfigured to operate in the simple mode for the remainder of the task. In spite of a missed checkpoint for a sub-task, we can guarantee safety for the entire task by reserving enough time between the checkpoint and the task's WCET to

1. reconfigure the pipeline to operate in simple mode,
2. complete the sub-task i whose checkpoint was missed (in simple mode) and
3. execute the remaining sub-tasks in simple mode.

Items 1 and 3 can be tightly bounded by considering the fixed overhead for mode switching of the architecture and by obtaining the sum of the WCETs of the remaining sub-tasks. Item 2, however, cannot be tightly bounded since we do not know the WCET of the *remaining* part of a sub-task in the middle of its execution. Nonetheless, a looser (and safe) bound is given by the WCET of the entire sub-task i. For now, we will express WCET in terms of cycles, referred to as worst-cases execution cycles (WCEC) in the following.

Checkpoints are derived from the WCECs of sub-tasks in the simple mode. The WCEC budget for each sub-task is depicted in Figure 2.

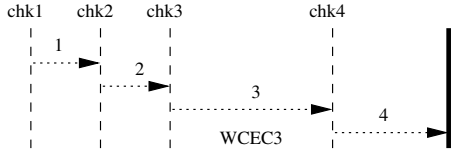


Fig. 2: WCECs of sub-tasks in simple mode.

Using the overall WCEC bound of a task and the corresponding bound for each sub-task i, $WCEC_i$, the checkpoint for sub-task i relative to the beginning of the task is

$$chk_i = WCEC - (switch\_delay + \sum_{k=i}^{s} WCEC_k) \quad (1)$$

For a sub-task i that misses its checkpoint, chk_i, while running in the complex mode: Equation 1 budgets enough time after the checkpoint to switch to the simple mode ($switch\_delay$), execute the unfinished sub-task i in the simple mode (even though part may have already executed in complex mode), and execute all remaining sub-tasks $i+1$ through s in the simple mode.

Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004)
1052-8725/04 $20.00 © 2004 IEEE

For the example task in Fig. 2, Equation 1 yields the corresponding checkpoints chk1-chk4. Note that the overall task's WCEC, not explicitly depicted in the figure, is at least as large as the cumulative sub-tasks' WCECs plus the modest overhead to switch the architectural mode (only several cycles to drain the pipeline and then reconfigure). If the task's WCEC is no larger than this minimum, speculation in the complex mode cannot be commenced immediately since the required "headstart" for gauging progress, described earlier in Sect. 2.1, is absent, *i.e.*, chk1 is reached immediately upon starting the task.

2.1.2 Detecting Missed Checkpoints

To detect missed checkpoints, progress of execution is gauged by a hardware watchdog counter, which can be queried for its value or set to a new start value through a memory-mapped location. The counter operates as a count-down mechanism, decremented on each cycle of execution, to alert the system when a checkpoint is missed, *i.e.*, if the counter reaches a zero value. Upon start of the first sub-task, the watchdog is initialized as

$$watchdog = chk_1.$$

This is depicted in Figure 3, which includes an explicit headstart amount to allow speculation in the complex mode to commence with the first sub-task. Derivation of the headstart amount is deferred to Section 3. The figure also shows sub-tasks' progress in the complex mode, *i.e.*, actual execution times. For any subsequent sub-task i, the watchdog counter is incremented by the WCEC between chk_{i-1} and chk_i:

$$watchdog = watchdog + chk_i - chk_{i-1}.$$

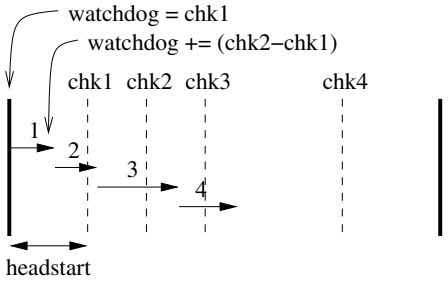


Fig. 3: Using watchdog to gauge progress in complex mode.

Notice that by adding to the watchdog before it expires in a previous sub-task, we gain slack for subsequent sub-tasks by being able to exploit left-over cycles from preceding ones. Hence, as execution progresses, sub-tasks are less likely to miss checkpoints due to operational properties of watchdog slack passing. Should the watchdog reach a zero value, due to it being decremented on each cycle of execution within the processor, it would indicate that a current sub-task missed its checkpoint. This raises a watchdog underflow exception. Such an exception causes the hardware to drain the pipeline and then invoke a light-weight exception handler in software that reconfigures the processor to stage subsequent execution in simple mode.

2.2 VISA System Design

Formally, the "virtual simple architecture" is the specification of a hypothetical simple pipeline, adhered to by both the static worst-case timing analysis tool (for generating WCETs) and the simple mode of the complex processor. The VISA specification is outlined in Fig. 4 (left-hand side). Fig. 4 also describes the complex processor used in this paper (right-hand side), which can be downgraded to the VISA-compliant simple mode of execution.

	VISA specification	**Complex Pipeline**
Pipeline Stages	fetch, decode, register read, execute, memory, writeback	fetch, dispatch, issue, register read, execute or agen/memory, writeback, retire
Fetch Unit	1 instr./cycle	4 instr./cycle
	static branch prediction: BT/FNT heuristic	dynamic branch prediction: 2^{16}-entry *gshare*, 16-bit BHR
Execution Core	in-order execution scalar execution (1 instr./cycle) 1 unpipelined universal function unit	out-of-order execution 4-way superscalar (4 instr./cycle) 4 pipelined universal function units 128-entry reorder buffer 64-entry issue queue 64-entry load/store queue 2 ports to load/store queue + DS
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

Fig. 4: Left: Virtual simple architecture specification. Right: Architecture of complex processor.

Embedding a simple mode of execution within a contemporary, high-performance pipeline does not incur significant hardware overhead or design complexity [1]. The simple mode is a subset of the existing pipeline and typically involves disabling complex features through simple gating. Simple modifications to key pipeline stages are described in the precursor architectural work [1]. For example: switching from dynamic to static branch prediction involves overriding the dynamic branch predictor's output with the static prediction, via a 1-bit 2:1 MUX; switching from superscalar to scalar execution involves disabling the unused superscalar ways, funneling instruction flow through only one of the ways; switching from out-of-order to in-order execution involves allowing only one instruction in the issue queue per cycle or bypassing it altogether; etc. To sum up, the simple mode of execution is not a separate component, rather, it is a downgrading of the existing complex pipeline and leverages the existing pipeline datapath.

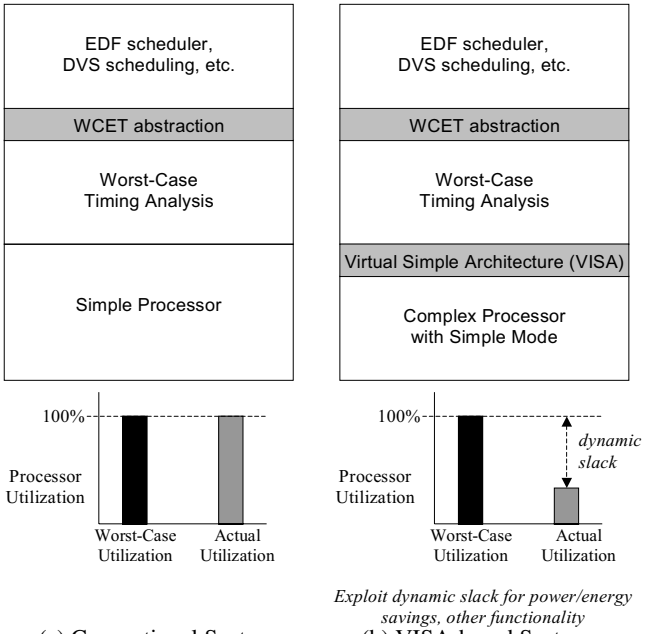
2.3 Benefits of VISA Framework

Fig. 5 captures the essence of the VISA concept and its benefits. It contrasts a *conventional real-time system* on the left with a *VISA-based system* on the right. The conventional system is limited to using an explicitly-safe, simple processor. In contrast, the VISA-based system can use a complex processor. The VISA safety framework ensures that the VISA-based system with complex processor and the conventional system with simple processor are provably equivalent in the worst case. In both cases, WCET analysis observes a simple processor. In the case of the VISA-based system, this abstraction is provided by the virtual simple architecture specification that sits above the complex processor, and guaranteed by the gauging mechanism plus the embedded simple mode of execution if needed. Thus, the first key benefit is providing a comparable worst-case performance bound for the complex processor without explicit analysis.

The second key benefit is that the complex processor typically introduces a substantial speed differential, resulting in the creation of large amounts of dynamic slack. This manifests itself by very early task completions, hence, low actual utilization of the complex processor as shown in the figure. The spare computational capacity can be used to increase functionality, *e.g.*, by scheduling additional soft-real-time or non-real-time tasks. Alternatively, dynamic slack can be used to reduce power via dynamic frequency/voltage scaling (DVS). Higher-level DVS scheduling techniques benefit trans-

Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004)
1052-8725/04 $20.00 © 2004 IEEE

parently from early task completions without specific knowledge of what causes the slack (in this case, speculative task execution on the complex processor).



(a) Conventional System, Simple Processor. (b) VISA-based System, Complex Processor.

Fig. 5: "Worst-case-equivalent" systems.

Note that the complex processor in a VISA-based system is intrinsically less power efficient than the simple processor used in a conventional system *for the same frequency/voltage*. This is evident from Fig. 6, based on detailed cycle-level architectural simulations augmented with Princeton's detailed Wattch power models [5]. (This detailed simulator is the basis for all results reported in this paper.) The first two bars in the graph show: (1) total energy consumption of the baseline simple processor used in this paper (scalar, in-order) running at the peak frequency of 1 GHz; (2) total energy consumption of the complex processor used in this paper (4-issue superscalar, out-of-order execution) running at the same frequency of 1 GHz. (The C-lab benchmark *adpcm* is used in these experiments. Wattch models are configured for aggressive clock gating of idle units.) Each bar is also annotated with the benchmark's execution time on the corresponding processor configuration. At the same frequency (1 GHz), the complex processor consumes 5% more energy than the simple processor. What this result overlooks is the fact that the complex processor finishes the task much faster, at the same frequency, due to significantly higher instruction-level parallelism (ILP) (0.68 ms vs. 2.37 ms). The complex processor's power efficiency advantage comes from parallelism. The frequency/voltage of the complex processor can be lowered with respect to the simple processor's frequency/voltage and offset with higher parallelism to yield roughly the same task execution time as the simple processor. The third bar in Fig. 6 shows total energy consumption of the complex processor at 300 MHz, the frequency which yields a similar task execution time as the simple processor at 1 GHz (2.14 ms vs. 2.37 ms). Notice the frequency/voltage reduction more than compensates for the complex processor's higher intrinsic power – yielding a net energy savings of 63%.

The graph in Fig. 6 also gives a coarse breakdown of where energy is being expended for each processor configuration. The breakdown helps explain why, even at the same frequency of 1 GHz, the initial energy disadvantage of the complex processor is not as significant as one would expect – only 5%. Energy is broken down into energy consumed in the execution core, the instruction and data caches (combined), and the clock tree. Aggressive clock gating ensures that idle units – unused ALUs, register file ports, cache ports, etc. – do not expend energy during any given cycle. Thus, ignoring wrong-path fetching and execution, energy consumed in the execution core and caches is primarily determined by the program, which is invariant. Therefore, aggressive clock gating has the effect of reducing the energy gap between the complex and simple processors. Nonetheless, the following contribute to more energy expended in the complex processor's core logic than the simple processor's core logic (3 mJ vs. 1.6 mJ): (1) corresponding structures (register file, etc.) in the complex processor's core are larger; (2) the complex processor has additional structures such as rename logic, dynamic scheduling logic, etc.; (3) a branch misprediction in the complex processor typically brings more incorrect instructions into the pipeline that are ultimately squashed, wasting energy. (On the other hand, the simple processor typically has more branch mispredictions.) The cache energy differential between the two processors is negligible due to the clock gating effect described above and the fact that the I-cache and D-cache configurations are the same for the two processors. Interestingly, it is clock-tree energy consumption that penalizes the simple processor. Although its clock tree is smaller (we assume the simple processor has 1/4 the die area, even with the two large caches) and, therefore, its clock-tree energy per clock tick is less, the benchmark takes 3.5 times more clock cycles to execute on the simple processor. The clock-tree energy builds up and ultimately exceeds the total clock-tree energy of the complex processor.

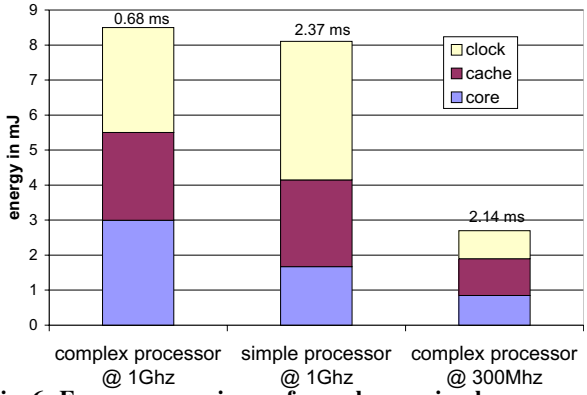


Fig. 6: Energy comparisons of complex vs. simple processors.

3. VISA SPECULATION OVERHEAD

As explained in Section 2.1, a headstart is needed to facilitate speculation. From a correctness standpoint, the headstart amount can be set arbitrarily.

At one extreme, using a headstart of zero cycles would cause the first sub-task to miss its checkpoint right away, since the initial value loaded into the watchdog counter is zero. In this case, the entire task is executed in the simple mode, failing to exploit the high-performing complex mode. More generally, using a headstart that is artificially low does not give the complex mode a reasonable chance to succeed. Missed checkpoints in this case are not actually due to anomalous, underperforming scenarios, but are rather due to an artificially low headstart amount.

On the other hand, using a headstart amount that is artificially in-

Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004)
1052-8725/04 $20.00 © 2004 IEEE

flated is also undesirable. It artificially inflates perceived worst-case utilization, possibly to the point of exceeding schedulability bounds.

When VISA-based speculation is combined with DVS scheduling, as discussed later, either scenario will result in higher energy consumption – due to running inefficiently in simple mode (after prematurely mispredicting) or due to exaggerating WCETs of future tasks, for under- and over-estimations, respectively. Thus, a good approximation is needed to select a headstart for scheduling.

3.1 Systematic Headstart Assessment

As mentioned above, the headstart amount is arbitrary from a safety standpoint. Our objective is to minimize the headstart amount while providing reasonable time for the complex mode to complete each sub-task, *i.e.*, to essentially eliminate mispredictions in the absence of any complex-mode performance anomalies. To this end, we profile the tasks on the complex mode. Let PEC_k denote the predicted execution cycles of sub-task k on the complex mode obtained by profiling. Recall that the timing-safe gauging method ensures that enough WCEC is budgeted after the checkpoint of each sub-task k for execution of the unfinished sub-task k and all remaining sub-tasks in simple mode, should the checkpoint be missed. This is shown in Fig. 7 for sub-task 3: WCECs of sub-tasks 3 and 4 are budgeted after sub-task 3's checkpoint, chk3. To make a miss of chk3 unlikely, we should anticipate an amount of time equal to the predicted (or profiled) execution time of sub-task 3 in the complex mode (PEC_3) prior to chk3. And we can assume, without restricting safety, that prior sub-tasks executed according to their profiled time as well (otherwise an earlier checkpoint would have missed prior to reaching this point). Hence, a good headstart amount δ_3 considering only sub-task 3 is given by the difference between chk1 (the original start boundary) and the new boundary as shown in Fig. 7. The same procedure is repeated for all sub-tasks individually, and the maximum headstart amount δ_k among all the sub-tasks is the desired headstart amount for the overall task.

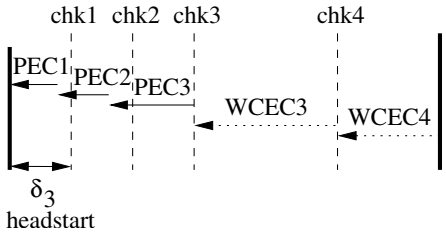


Fig. 7: Headstart amount, considering only the third sub-task.

Based on Fig. 7, the total execution time for a task on the VISA framework, assuming sub-task k misses its checkpoint, is

$$(\sum_{i=1}^{k-1} PEC_i + PEC_k + WCEC_k + \sum_{i=k+1}^{s} WCEC_i) \quad (2)$$

Combining the PEC terms together and the WCEC terms together, equation 2 can be simplified as

$$(\sum_{i=1}^{k} PEC_i + \sum_{i=k}^{s} WCEC_i) \quad (3)$$

The minimum headstart (δ_k) required for sub-task k to not miss its checkpoint (assuming no anomalies) would then be the difference between the execution time on the VISA framework (denoted by equation 3) and the original worst-case execution time, which is the sum of the WCECs of the individual sub-tasks.

$$\begin{aligned} \delta_k = \; & (\textstyle\sum_{i=1}^{k} PEC_i + \sum_{i=k}^{s} WCEC_i) - \sum_{i=1}^{s} WCEC_i \quad (4) \\ & = PEC_k - \textstyle\sum_{i=1}^{k-1} (WCEC_i - PEC_i) \end{aligned}$$

The latter (rewritten) form of Equation 4 above is intuitive. Essentially, we expect the needed headstart for sub-task k to be at least as large as its predicted execution time in the complex mode, PEC_k. However, this amount can be lessened by slack gained due to early completion of prior sub-tasks, caused by the difference between $WCEC_i$ and PEC_i of all prior sub-tasks i.

To lower the chance that *any* checkpoints are missed, the overall headstart δ for a task should be the maximum of the headstarts derived for *all* of its sub-tasks:

$$\delta = \max_{1 \le k \le s} (\delta_k) \quad (5)$$

3.2 Static Padding of Headstart

To ensure that a sufficient headstart is available to confidently initiate any task in complex mode, we *explicitly pad* each task's WCEC with its headstart amount. Hence, this technique is called the *explicit padding* approach.

A task's WCEC provided to the scheduler for the *explicit padding* approach is the sum of the individual sub-tasks' WCECs plus the headstart term δ, as detailed below:

$$WCEC_{pad} = \delta + \sum_{i=1}^{s} WCEC_i \quad (6)$$

Substituting for δ using equations 5 and 4, we get the expression for the padded WCEC as:

$$\begin{aligned} WCEC_{pad} = \; & \max_{1 \le k \le s} (\delta_k) + \textstyle\sum_{i=1}^{s} WCEC_i = \quad (7) \\ & \max_{1 \le k \le s} (\textstyle\sum_{i=1}^{k} PEC_i + \sum_{i=k}^{s} WCEC_i) \end{aligned}$$

Notice that this padding amount avoids unnecessary switches from complex to simple mode, which impacts the execution efficiency. It does not impact the correctness of the scheme as paddings that are too tight only result in simple execution for the remainder of a task after a checkpoint is missed. The overall deadline is always met.

The advantage of the *explicit padding* approach is that it allows each task to be started speculatively in complex mode because the headstart amount is automatically budgeted in each task's WCET. However, explicitly padding the WCET with the headstart inflates the overall worst-case utilization. Although the overhead tends to be small, as we will show in the results section, task-sets with near-100% worst-case utilization may not be able to accommodate the static headstart padding, precluding speculation in the complex mode altogether, thus defaulting to only the simple mode. Accordingly, in the next section, we propose a novel zero-overhead approach called *dynamic headstart accrual* to eliminate the need for padding tasks' WCETs.

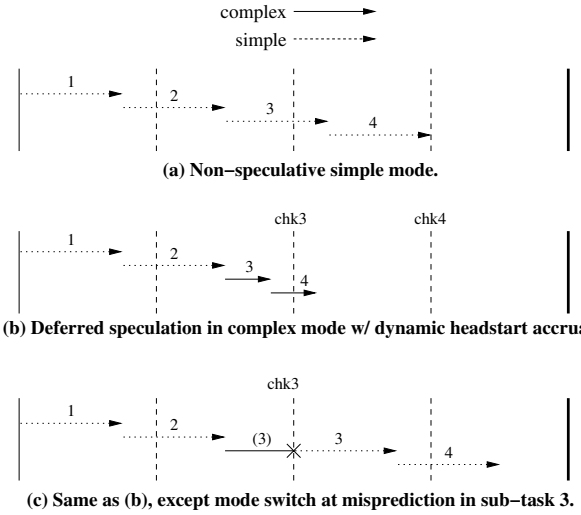
4. DYNAMIC HEADSTART ACCRUAL

We propose a novel VISA speculation approach, in which tasks' WCETs do not have to be padded with the headstart amounts. In this approach, a task is initiated in the simple mode, contrary to the *explicit padding* approach (which immediately begins speculation in the complex mode, enabled by headstart padding). Because actual execution times are typically less than WCETs even in the simple mode, dynamic slack is generated naturally (albeit more slowly

Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004)
1052-8725/04 $20.00 © 2004 IEEE

than for the complex mode). This natural slack can be gathered for eventually enabling the complex mode for a future sub-task. Since the required headstart is dynamically accrued, we refer to this technique as *dynamic headstart accrual*.

Fig. 8(a)-(c) illustrates the new approach. Fig. 8(a) shows non-speculative (conservative) execution in the simple mode. Fig. 8(b) shows execution on a VISA-protected processor employing dynamic headstart accrual. Notice the task is started in the simple mode (denoted by dashed arrows). At the end of the second sub-task, sufficient dynamic slack has accrued to enable switching to the complex mode. Hence, the remaining portion of the task is executed in complex mode (denoted by solid horizontal arrows). Fig. 8(c) shows the case where the pipeline is reconfigured to complex mode at the end of sub-task 2 (due to sufficient headstart accrued), but sub-task 3 misses its checkpoint (complex-mode underperformance anomaly). We see that the contractual WCET is still guaranteed despite the missed checkpoint because of the time budgeted after the checkpoint to complete sub-task 3 and remaining sub-tasks in the simple mode.



(a) Non−speculative simple mode.

(b) Deferred speculation in complex mode w/ dynamic headstart accrual.

(c) Same as (b), except mode switch at misprediction in sub−task 3.

Fig. 8: Illustration of deferred speculation.

With this new approach, no padding is required since the headstart is dynamically accrued. Thus, a task's WCET is simply that of Equation 6 with zero padding, *i.e.*, $\delta = 0$. Now, the watchdog is leveraged for a different purpose, *i.e.*, to measure the amount of slack accrued while in the simple mode. The watchdog is initialized to zero and, at the start of each sub-task, the watchdog is incremented by the sub-task's WCEC. Although the watchdog is decremented every cycle, as before, it will never go below zero in the safe simple mode, and any amount that remains after the completion of a sub-task reveals the accrued slack up to that point.

Based on early sub-task completion, one can determine after how many sub-tasks a change from simple mode to complex mode becomes beneficial in the sense that future checkpoints are unlikely to be missed.

This is exactly the same goal as for the explicit padding approach, only now we are waiting for the desired headstart amount δ to accrue naturally. Before starting a new sub-task x, we determine if it is beneficial to switch to the complex mode by comparing the amount of available slack in the watchdog to a headstart amount δ; this δ is derived using the same procedure as before, only now we do not need to consider prior completed sub-tasks when finding the maximum. Hence, the condition for mode switching at the beginning of sub-task x is:

$$watchdog \geq \max_{x \leq k \leq s} (\delta_k) \tag{8}$$

The δ_k amounts can be computed *a priori* since they only depend on PECs and WCECs. They can be stored in a lookup table. Hence, only a small, constant overhead is incurred in looking up the appropriate δ_k's and comparing their maximum with the watchdog counter.

Notice that the watchdog does *not* need to be updated when switching from simple mode to complex mode, since it conveniently contains the initial headstart. If the condition represented by Inequation 8 is satisfied at the beginning of some sub-task, only a mode switch (to complex) occurs.

Slack in the watchdog can be due to early completion of initial sub-tasks up to $x-1$, *i.e.*, it can be *intra-task slack* accumulating in the watchdog. This condition would need to be evaluated at each new sub-task x until it is satisfied and the mode is switched to complex. There is no need to check the condition after mode switching as progress is then gauged via checkpoints as before.

The condition can be weakened by considering *inter-task slack*, both of static and dynamic nature. Inter-task slack is discovered at different rates depending on the real-time scheduling policy and can be used within the VISA framework. Such slack would already be available for the first sub-task, which may allow us to immediately switch to the complex mode even though no explicit padding was imposed. This is discussed in Section 5.

After switching to complex mode, execution will only revert to simple mode if a checkpoint is missed. The dynamic accrual scheme allows us to eventually revert to complex mode again once enough intra-task slack has accumulated, another strength of this approach.

The primary advantage of the dynamic headstart accrual approach is that WCETs need not be padded and, hence, the overall worst-case utilization is unaffected. Hence, any task-set that can be scheduled on the hypothetical simple pipeline can be scheduled on a VISA-protected processor. An additional benefit is that the dynamic accrual scheme supports multiple switches from simple mode to complex mode provided sufficient slack has accrued.

The drawback of this scheme is its dependence on the creation of dynamic slack to switch to complex mode. If accrued slack is insufficient, the task will be perpetually executed in simple mode. So, the dynamic headstart accrual approach is well suited to task-sets that have high worst-case utilization but lower actual utilization on the hypothetical simple pipeline.

5. TRANSPARENT DVS SCHEDULING

Let us consider the implications of dynamic frequency/voltage scaling (DVS) on top of the VISA framework. The motivation for DVS on top of the VISA framework is given by, on average, much earlier completion of tasks in the complex mode. In the following, we discuss transparency of the VISA framework in this context.

Transparency stems from the fact that the VISA framework expresses execution budgets in cycles, and, therefore, is independent of frequency changes and corresponding changes in execution time. In particular, a task's WCEC, headstart amount, and checkpoints are all expressed in cycles. Whereas DVS lowers (raises) frequency to extend (compress) execution time, cycles are invariant. As such, the VISA framework functions correctly regardless of the safe fre-

Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004)
1052-8725/04 $20.00 © 2004 IEEE

quency selected by the DVS scheduling algorithm.

Likewise, DVS scheduling algorithms are oblivious of the underlying VISA framework. DVS may observe much more slack than usual, due to early completion of tasks in the complex mode. This slack is exploitable without specific knowledge of how the slack accrued (in this case, fewer execution cycles due to high instruction-level parallelism in the complex mode).

The VISA framework does not limit itself to any specific real-time DVS scheduling algorithm. DVS scheduling can be applied to simple and complex modes alike. Static schemes may choose a uniform frequency over all tasks while dynamic schemes may vary frequencies between tasks, different jobs of the same task or even within a job at preemption points.

The VISA framework can also benefit from real-time DVS scheduling algorithms in a semi-transparent manner. This is of particular interest for *dynamic headstart accrual*, *i.e.*, in the absence of padding. Recall that execution had to start in the simple mode since no intra-task slack was available prior to the first sub-task. If, however, a real-time DVS scheduler discovers slack as an inherent part of the scheduler invocation, this slack can be exploited to *immediately* start a task in the complex mode.

Consider the Look-Ahead DVS algorithm by Pillai and Shin [24]. It determines the amount of slack s available for the current task, which can be fully exploited for frequency scaling without risking a missed deadline. Using the condition represented in Inequation 8, only now checking if the slack s is greater than the needed δ, the VISA framework may be able to switch to the complex mode as soon as the first sub-task. Hence, a real-time DVS scheduler may choose to apportion some of the slack s to the watchdog, slightly reducing the amount of slack available for frequency scaling (from s to s', by δ from Inequation 8) but, in return, enabling a task to immediately commence in complex mode:

$$s' = s - \delta$$

$$watchdog = \delta$$

Above, we claimed that the VISA framework is independent of processor frequency since it expresses execution budgets in cycles. This claim holds under linear scaling of execution time with processor frequency. Although execution time does not scale linearly with processor frequency due to constant-time memory accesses, it is safe to assume linear scaling, albeit, the resulting WCET may not be as tight since one has to assume the memory latency (in cycles) at the maximum frequency for all lower frequencies. This problem is orthogonal to the VISA framework and can be addressed by augmenting timing predictions with memory modeling, such as given by FAST [27] or by using per-frequency WCETs [26, 1]. Using this conservative assumption about linear scalability, we can safely (but not necessarily tightly) assert the WCET of a task as

$$WCET = \frac{WCEC}{f} \tag{9}$$

given the WCEC at the maximum processor frequency f. This $WCET$ is then used by DVS scheduling algorithms as a base for scaling frequency and can be easily determined for either *explicit padding* or *dynamic headstart accrual*, using Equation 9.

6. PREEMPTIONS

For a set of real-time tasks under preemptive scheduling, the effect of suspending and resuming tasks has to be considered. We discuss two aspects related to preemptions. First, we discuss how the integrity of the watchdog checking mechanism is preserved in the presence of preemptions. Second, we discuss modeling of preemption overheads, which applies to both VISA and non-VISA real-time systems.

6.1 Saving/Restoring the VISA State

For the *explicit padding* approach, supporting preemptions in hardware is similar to precise handling of exceptions/interrupts: The state used to manage a task in the VISA framework is saved as part of the task's context and later restored. The following state is included in the context:

- The current pipeline mode (complex or simple) and
- the content of the watchdog counter.

When a task is resumed, the pipeline mode is restored to that prior to preemption, which may differ from the mode of the preempting task. Likewise, the watchdog counter is restored to its previous value and used to monitor further progress of the interrupted sub-task. (However, as usual, the watchdog counter is not used if the restored mode is the simple mode, since progress is not gauged in simple mode.)

The technique for saving/restoring the watchdog counter is independent of frequency changes introduced by transparent DVS scheduling, since execution budgets are expressed in cycles. Decoupling of the VISA framework from frequency variations was discussed in Section 5.

For the *dynamic headstart accrual* scheme, the pipeline mode and watchdog counter do not have to be saved/restored when the task is preempted/resumed. We exploit the approach described in Section 5, whereby the DVS scheduler re-initializes the VISA state upon resuming a preempted task. The watchdog counter is re-initialized on the basis of slack s that has accumulated since the task was preempted. Likewise, the pipeline is configured in either the simple or complex mode on the basis of the result of testing the condition in Inequation 8.

6.2 Accounting for Preemption Overheads

In multi-tasking real-time systems, the worst-case execution time of the task scheduler itself must be accounted for in schedulability analysis. Our scheduler's WCET includes selecting the next task based on deadline scheduling, such as rate-monotone or EDF scheduling, applying a DVS scheduling algorithm, and saving/restoring register contexts in the case of preemptions. Safely accommodating scheduler overheads is independent of VISA, *i.e.*, these overheads must be considered for VISA and non-VISA real-time systems alike. Nonetheless, there are several alternatives for accounting for the WCET of the scheduler.

We present an approach very convenient in two respects:

1. It implicitly accounts for the worst-case number of scheduler invocations (bounded by two per task, one for when the task is released and one for when it completes).
2. It enables the task scheduler to execute in complex mode without first delineating it into sub-tasks, setting interim checkpoints, etc., as is done with other tasks.

Our solution is to consider the task scheduler to be part of the first sub-task and the last sub-task of each task in the task-set. The reason the scheduler does not need to be partitioned is that, effectively, it is no longer a task, but rather a part of a sub-task in another task. To summarize, the overhead of the task scheduler is accommodated

by padding the WCECs of the first and last sub-tasks (for each task in the task-set), as follows:

$$WCEC_i = WCEC'_i + WCEC_{scheduler} \; for \; i = 1, s \quad (10)$$

$WCEC'_i$ is the original WCEC of the ith sub-task before padding it with the scheduler overhead.

7. EXPERIMENTAL FRAMEWORK

We simulate a multi-tasking real-time system (including periodic task-sets, EDF scheduler, and real-time DVS scheduler) on a detailed cycle-level architectural simulator. The architectural simulator was custom-built using the Simplescalar toolset [6]. We integrated the Wattch power models [5] into the simulator, elaborated below. Tasks and the scheduler were compiled using the gcc-based Simplescalar compiler, which targets the PISA instruction set (MIPS-like).

We use the Look-Ahead DVS real-time scheduling algorithm for scheduling tasks and choosing a safe processor frequency [24]. In our experiments, we investigate two processor models:

1. *explicitly-safe simple*: This represents a baseline, explicitly-safe simple processor that directly implements the VISA pipeline specification. Recall, the VISA pipeline specification was reviewed in Fig. 4.

2. *VISA-protected complex*: This represents a VISA-protected complex processor, that supports the complex and simple modes. The microarchitecture of the complex processor was also reviewed in Fig. 4.

For *VISA-protected complex*, we study both the *explicit padding* and *dynamic headstart accrual* speculation approaches. Recall from Section 4 for the *dynamic headstart accrual* approach, we can switch from simple mode to complex mode as soon as the accrued slack is greater than the headstart (δ) needed for speculation. As the headstart amount is affected by the sizes of sub-tasks, we also investigate the impact of coarse-grained *vs.* fine-grained sub-task selection.

7.1 Power Modeling

The Wattch power models were modified to closely resemble the microarchitecture of contemporary superscalar processors. Instead of the original RUU-based microarchitecture modeled by Wattch, we consider a separate physical register file, active list, issue queue, and load/store queue. To support dynamic voltage scaling (DVS), we built in support for an extrapolated Intel XScale DVS model with 37 frequency/voltage pairs in range of 100MHz/0.70V to 1GHz/1.8V [13].

During idle time, we operate at the minimum frequency/voltage since the overhead of switching into sleep modes is prohibitive on realistic architectures in the presence of real-time task-sets with periods as short as 10 ms.

Since *explicitly-safe simple* directly implements the VISA pipeline specification, its microarchitectural components (e.g., register file) are sized accordingly. Hence, it is more power-efficient than the simple mode of *VISA-protected complex*, which must still access larger microarchitectural components of the complex pipeline.

7.2 Static Worst-Case Timing Analysis

We use our static timing tool to generate WCETs on a sub-task basis. Analysis targets the VISA pipeline specification that was reviewed in Fig. 4. Static timing analysis includes static instruction cache simulation, pipeline simulation and path analysis for alternate paths in the control-flow. Interprocedural control-flow is analyzed in a bottom-up fashion to derive worst-case execution times (or cycles) for code portions (in our case sub-tasks) or even entire tasks (for schedulability). The effect of data caching was simulated for a cache size sufficiently large (64KB) to incur only cold misses, which allowed us to tightly bound the WCET with our tools. Other approaches to ensure predictability for the data cache, such as those based on locking caches [32], could be used in conjunction with the VISA framework to handle smaller cache sizes. The details of static timing analysis [11] and its adaptation to Simplescalar [1, 27] are beyond the scope of this paper.

7.3 Benchmarks and Task-Sets

The benchmarks are a subset of the C-lab real-time benchmark suite [8]. These benchmarks are widely used in timing analysis. We analyze fully optimized (gcc -O3) code generated for these benchmarks. Table 1 depicts two different sub-task selections per benchmark in the second and third columns: fine-grained and coarse-grained, where the former has twice the number of sub-tasks of the latter. The fourth and fifth columns indicate the WCET and the padded WCET for the *explicit padding* approach, respectively. Since the *dynamic headstart accrual* approach does not need explicit padding, it uses the unpadded WCETs, as does *explicitly-safe simple*. The final two columns indicate actual execution times at the maximum frequency for the *explicitly-safe simple* processor (also the simple mode) and the *VISA-protected complex* processor.

Tab. 1: Benchmarks.

bench-mark	# sub-tasks		WCET (ms)	Padded WCET (ms)	Average execution time (ms) @ 1 GHz	
	fi ne-grain	coarse-grain			*explicitly-safe simple*	*VISA-protected complex*
adpcm	16	8	3.35	3.46	2.43	0.64
cnt	10	5	0.16	0.17	0.07	0.02
fft	10	5	0.59	0.62	0.36	0.06
lms	20	10	0.19	0.195	0.17	0.04
mm	20	10	2.25	2.35	2.10	0.66
srt	20	10	3.53	3.65	1.82	0.55

We created various task-sets by combining the benchmarks above. Each task-set has three tasks whose periods are indicated in Table 2. For example, task-set LCF is composed of lms, cnt, and fft, with periods of 0.6 ms, 0.52 ms, and 2.0 ms, respectively. Task-sets were designed to obtain various mixes of small and large tasks. There is one task-set composed of all small tasks (LCF), one task-set composed of all large tasks (AMS), four task-sets composed of one large and two small tasks (LFM, CLS, LMC, ACL), and four task-sets composed of one small and two large tasks (ASL, ASF, MSL, ASC).

For the first set of experiments, task periods were randomly selected while ensuring that the task-set is schedulable by both *explicitly-safe simple* and *VISA-protected complex* with *explicit padding*. Notice that *explicitly-safe simple* does not require any padding and, hence, has lower utilization than *VISA-protected complex* as shown in the last two columns in Table 2. Lower utilization allows DVS scheduling algorithms to exploit additional static slack whereas there is comparatively less static slack for *VISA-protected complex* with *explicit padding*.

To further demonstrate the benefits yielded by the *dynamic headstart accrual* approach, we construct additional task-sets (similar in composition to the ones in Table 2) such that the utilization is 1.0 based on unpadded WCETs. The first column of Table 3 indicates

Tab. 2: Task-sets. (Utilization less than 1.)

task-set	$task_1$ P_1 (ms)	$task_2$ P_2(ms)	$task_3$ P_3(ms)	U_{max} *explicitly -safe simple*	U_{max} *VISA-protected complex*
LCF	0.6	0.52	2.0	0.91	0.96
CLS	0.52	6.0	11.0	0.94	0.98
LMC	0.6	7.5	0.52	0.91	0.97
ACL	11.0	0.52	6.0	0.92	0.97
LFM	0.6	2.0	7.5	0.91	0.94
MSL	7.5	11.0	0.6	0.93	0.97
ASL	11.0	11.0	0.6	0.94	0.97
ASC	11.0	11.0	0.52	0.93	0.98
ASF	11.0	11.0	2.0	0.92	0.95
AMS	11.0	7.5	11.0	0.92	0.96

Tab. 3: Task-sets. (Utilization equals 1.)

task-set	$task_1$ P_1 (ms)	$task_2$ P_2(ms)	$task_3$ P_3(ms)	U_{max}
LCF	0.55	0.48	1.8	1.0
CLS	0.46	0.56	10.7	1.0
LMC	0.55	6.80	0.48	1.0
ACL	9.84	0.48	0.58	1.0
LFM	0.55	1.80	6.81	1.0
MSL	6.61	10.70	0.56	1.0
ASL	9.84	10.70	0.56	1.0
ASC	9.84	10.70	0.48	1.0
ASF	9.84	10.70	1.8	1.0
AMS	9.84	6.81	10.70	1.0

the name of the task-set while the next three columns show the periods of the member tasks. Notice that the periods here are smaller compared to the periods shown in Table 2.

In the experiments that follow, each task-set is simulated for 50 ms.

8. RESULTS

We now present the energy savings yielded by the VISA framework. Figure 9 shows the energy savings of *VISA-protected complex* relative to *explicitly-safe simple*, for both the *explicit padding* and *dynamic headstart accrual* approaches. Coarse-grained sub-tasks are used. *VISA-protected complex* yields energy savings ranging from 24% (AMS) to 58% (LFM). This is due to the fact that *explicitly-safe simple* runs at an average frequency of about 700MHz to 850MHz (depending on the task-set), as shown in Figure 10. On the other hand, Figure 10 shows that *VISA-protected complex* runs at a lower average frequency, ranging from about 275MHz to 450MHz (depending on the task-set and speculation approach).

In these experiments, no checkpoints were missed in the case of *VISA-protected complex*. This means that all tasks were executed using the complex mode, and the simple mode was not used at all. Though the complex mode is highly reliable, it is not provably safe, hence, the simple mode is needed to ensure that the WCET abstraction is preserved in unusual cases where the complex mode underperforms.

Notice that the *dynamic headstart accrual* approach yields more energy savings than the *explicit padding* approach. *Explicit padding* causes the DVS algorithm to use padded WCETs for all future tasks when calculating the frequency. On the other hand, *dynamic headstart accrual* causes the DVS algorithm to use unpadded WCETs for all future tasks plus the headstart amount for the next task when calculating the frequency. Because the perceived worst-case amount of work is less for the *dynamic headstart accrual* approach, it yields a lower average frequency than the *explicit padding* approach, as shown in Figure 10.

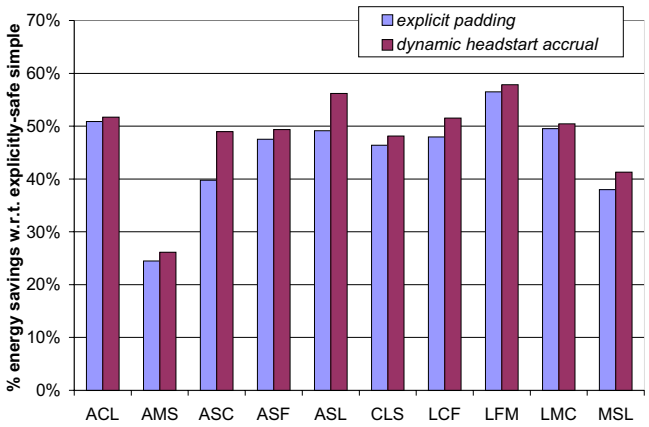


Fig. 9: Energy savings of *VISA-protected complex* with *explicit padding* and *dynamic headstart accrual* over *explicitly-safe simple*.

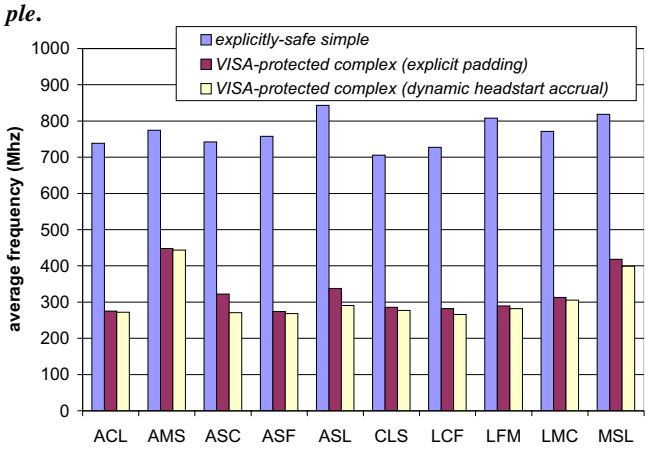


Fig. 10: Average frequencies of *explicitly-safe simple* and *VISA-protected complex* with *explicit padding* and *dynamic headstart accrual*.

Next, we present the energy savings yielded by *VISA-protected complex* for a task-set whose worst-case utilization (based on unpadded WCETs) is explicitly set to 1, *i.e.*, task periods are set as shown in Table 3. Since the tight schedule cannot accommodate padding, the *explicit padding* approach provides little benefit (the simple mode would be used perpetually). However, this task-set can certainly benefit from the *dynamic headstart accrual* approach.

Recall that the pipeline can be reconfigured from simple mode to complex mode as soon as the accrued slack is greater than the headstart amount (δ). So, a smaller headstart amount is desirable because it increases the chances and speed with which speculation is engaged. The headstart amount is less for tasks that are partitioned more finely (due to smaller PECs), so we also study the impact of coarse-grained *v.s.* fine-grained sub-task selection (refer to Table 1).

Figure 11 shows the energy savings yielded by *VISA-protected complex* employing the *dynamic headstart accrual* approach, for both coarse-grain and fine-grain sub-tasks. We see that *VISA-protected complex* yields 19% to 58% energy savings relative to *explicitly-safe simple*. Also notice that, with the exception of one

Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004)
1052-8725/04 $20.00 © 2004 IEEE

task-set (ASL), fine-grained sub-task selection yields higher energy savings than coarse-grained sub-task selection. Figure 12 shows the average frequencies for *explicitly-safe simple* and for *VISA-protected complex*, the latter with both coarse-grain and fine-grain sub-tasks. As observed previously, the average frequencies for *VISA-protected complex* (275MHz to 500MHz) are always much lower than the average frequencies for *explicitly-safe simple* (750MHz to 875MHz). For *VISA-protected complex*, the first task in a hyperperiod starts in simple mode and the pipeline is later reconfigured to operate in complex mode when enough slack has accrued. Typically, the switch to complex mode occurs in the middle of the first task (due to intra-task slack) or at the end of the first task (due to inter-task slack). Subsequent tasks typically find enough slack to start in complex mode due to the slack provided by the DVS algorithm (see Section 5).

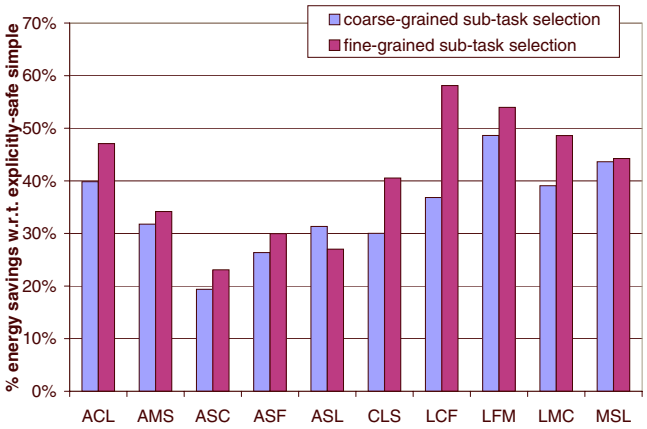


Fig. 11: Energy savings of *VISA-protected complex* using *dynamic headstart accrual* relative to *explicitly-safe simple*, for task-sets with worst-case utilization of 1.

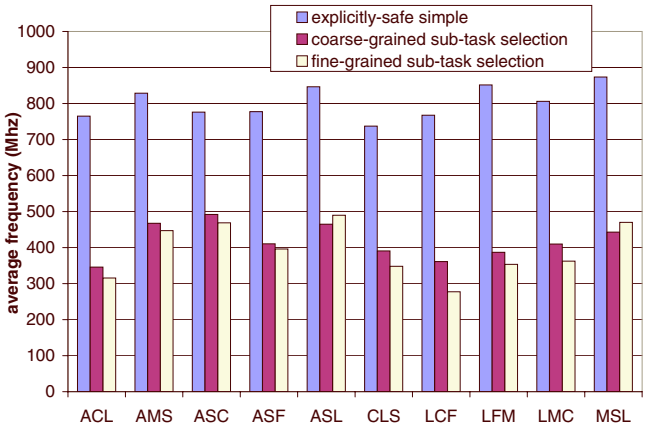


Fig. 12: Average frequencies of *explicitly-safe simple* and *VISA-protected complex* using *dynamic headstart accrual*, for task-sets with worst-case utilization of 1.

9. RELATED WORK

Scheduling theory for hard real-time systems, such as rate-monotone and EDF scheduling, relies on known WCET bounds for tasks [20, 7, 30]. For modern architectures, timing bounds on a task's WCET are increasingly difficult to obtain as the complexity of architectures increases [10].

Architectural support for hard real-time systems was proposed as cache partitioning to separate the cache footprint of each real-time task [16]. Subsequently, software partitioning schemes of caches were studied [33, 18]. Our work differs in that VISA provides two execution modes, which combines maximum opportunities for execution speed with a fall-back mode for guaranteeing hard real-time deadlines.

Various methods to assess the WCET of tasks statically have been proposed in the past, ranging from processor models to caches and advanced architectural features, such as branch prediction and locking caches [23, 25, 34, 31, 22, 15, 17, 11, 9, 32]. While these techniques rely on static analysis of programs, recent work is taking a probabilistic approach to provide certainty levels of the WCET based on a sample of observed executions supported by the pWCET tool [4]. Any of these approaches can be combined with the VISA specification to obtain WCET bounds for the simple mode. The VISA framework then provides safety by gauging progress in complex mode since none of the above static analysis techniques can fully capture the effects of a complex architecture. Even the pWCET results at some certainty level can profit from the additional safety provided by VISA.

Related work on DVS scheduling for hard real-time systems has demonstrated significant energy savings for time-constrained embedded systems [29, 24, 3, 3, 14, 35]. These techniques generally are variations on slack reclamation schemes ranging from static to dynamic analysis schemes, with sources of slack originating from idle, inter-task savings (early completion) and intra-task savings (early sub-task completion). While we only demonstrated the applicability to Pillai's Look-Ahead DVS scheduling, VISA allows the integration of arbitrary DVS scheduling algorithms.

Hughes *et al.* [12] compared energy savings of architectural adaptation, frequency/voltage scaling, and combinations of the two in soft real-time systems. Interestingly, they found that complex microarchitectures are often more energy-efficient than simpler architectures under DVS. These gains result from matching performance at a lower frequency and were a motivating factor for exploiting slack for power savings in this paper. Single-task results are reported by Anantaraman *et al.* [1]. Our paper rigorously extends this approach to multi-tasking systems.

10. CONCLUSION

The VISA framework fundamentally transforms the way we view the provisioning of real-time guarantees. Sharing the burden of WCET enforcement between hardware (complex processor with downgraded simple mode) and software (analytical framework for checkpoints, headstarts, etc.), coupled with the abstraction of a virtual simple architecture, enables the enforcement of real-time schedules on contemporary processors.

In this paper, we rigorously extended the VISA framework to arbitrary multi-tasking systems, including integrity of the intra-task gauging mechanism with inter-task preemptions, transparency of the VISA framework to arbitrary RTOS software components, and optimality of VISA speculation overheads. A new VISA speculation approach was proposed, much more fluid in its mode-switching than the previous approach, that has zero speculation overhead.

Overall, this paper unveils the full extent of VISA's powerful abstraction capability, helping to modernize hard real-time systems by safely integrating contemporary processors.

Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004)
1052-8725/04 $20.00 © 2004 IEEE

Acknowledgements

Static timing analysis results were, in part, obtained by Sibin Mohan. This research was supported in part by NSF grants CCR-0207785, CCR-0208581, and CCR-0310860, NSF CAREER grant CCR-0092832, and generous funding and equipment donations from Intel, Microsoft and Ubicom. This paper does not necessarily reflect or represent the views of Qualcomm, Inc.

11. 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. In *30th Int'l Symp. on Computer Architecture*, pages 250–261, June 2003.

[2] A. Anantaraman, K. Seth, E. Rotenberg, and F. Mueller. Exploiting visa for higher concurrency in safe real-time systems. Technical Report TR-2004-15, Dept. of Computer Science, NC State Univ., May 2004.

[3] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Dynamic and agressive scheduling techniques for power-aware real-time systems. In *IEEE Real-Time Systems Symp.*, Dec. 2001.

[4] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *IEEE Real-Time Systems Symp.*, Dec. 2002.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th Int'l Symp. on Computer Architecture*, pages 83–94, June 2000.

[6] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Technical Report CS-TR-96-1308, Univ. of Wisc - Madison, CS Dept., July 1996.

[7] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer, 1997.

[8] C-Lab. Wcet benchmarks. Available from http://www.c-lab.de/home/en/download.html.

[9] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3):249–174, 2001.

[10] T. Hand. Real-time systems need predictability. *Computer Design (RISC Supplement)*, pages 57–59, Aug. 1989.

[11] 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.

[12] C. J. Hughes, J. Srinivasan, and S. V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *34th Int'l Symp. on Microarchitecture*, Dec. 2001.

[13] Intel. *Intel XScale Microarch. Tech. Summary*, July 2000.

[14] D. Kang, S. Crago, and J. Suh. A fast resource synthesis technique for energy-efficient real-time systems. In *IEEE Real-Time Systems Symp.*, Dec. 2002.

[15] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Embedded Tech. and Applications Symp.*, June 1996.

[16] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *IEEE Real-Time Systems Symp.*, pages 229–237, Dec. 1989.

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

[18] J. Liedke, H. Härtig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *IEEE Real-Time Embedded Tech. and Applications Symp.*, pages 213–223, June 1997.

[19] 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. In *IEEE Real-Time Systems Symp.*, pages 97–108, Dec. 1994.

[20] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Assoc. for Computing Machinery*, 20(1):46–61, Jan. 1973.

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

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

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

[24] P. Pillai and K. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Symposium on Operating Systems Principles*, 2001.

[25] P. Puschner. Computing maximum task execution times – a graph-based approach. *Real-Time Systems*, 9(4):(to appear), Oct. 1997.

[26] E. Rotenberg. Using variable-Mhz microprocessors to efficiently handle uncertainty in real-time systems. In *34th Int'l Symp. on Microarchitecture*, pages 28 – 39, Dec. 2001.

[27] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. Fast: Frequency-aware static timing analysis. In *IEEE Real-Time Systems Symp.*, pages 40–51, Dec. 2003.

[28] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. Real-time scheduling for a virtual simple architecture (visa). In *Work in Progress at IEEE Real-Time Systems Symp.*, pages 129–132, Dec. 2003.

[29] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Int'l Conf. on Computer-Aided Design*, 2000.

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

[31] H. Theiling and C. Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. In *IEEE Real-Time Systems Symp.*, pages 144–153, Dec. 1998.

[32] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *IEEE Real-Time Systems Symp.*, Dec. 2003.

[33] A. Wolfe. Software-based cache partitioning for real-time applications. In *Workshop on Responsive Computer Systems*, 1993.

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

[35] Y. Zhu and F. Mueller. Feedback edf scheduling exploiting dynamic voltage scaling. In *IEEE Real-Time Embedded Tech. and Applications Symp.*, pages 84–93, May 2004.