Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems

Aravindh Anantaraman*, Kiran Seth*, Kaustubh Patil†, Eric Rotenberg*, Frank Mueller†
ECE * and CSC † Departments, Center for Embedded Systems Research, North Carolina State Univ.
{avananta, krseth, ericro}@ece.ncsu.edu, {kspatil, mueller}@cs.ncsu.edu

Abstract

Meeting deadlines is a key requirement in safe real-time systems. Worst-case execution times (WCET) of tasks are needed for safe planning. Contemporary worst-case timing analysis tools can safely and tightly bound execution time on in-order single-issue pipelines with caches and static branch prediction. However, this simple pipeline appears to be a complexity limit, due to the need for analyzability. This excludes a whole class of high-performance processors from many embedded systems.

We reconcile the complexity/safety trade-off by decoupling worst-case timing analysis from the processor implementation, through a virtual simple architecture (VISA). A VISA is the timing specification of a hypothetical simple pipeline and is the basis for worst-case timing analysis. However, the underlying microarchitecture can be arbitrarily complex. A task is divided into multiple sub-tasks which provide a means to gauge progress on the complex pipeline. Each sub-task is assigned an interim deadline, or checkpoint, based on the latest allowable completion time of the sub-task on the hypothetical simple pipeline. If no checkpoints are missed, then the complex pipeline is as timely as the safe pipeline. If a checkpoint is missed, the pipeline switches to a simple mode of operation that directly implements the VISA so that execution time of unfinished sub-tasks is safely bounded. The significance of our approach is that we circumvent worst-case timing analysis of the complex pipeline, by dynamically confirming its behavior is bounded by worst-case timing analysis of a simpler proxy pipeline.

The benefit of using a high-performance processor is that tasks finish much sooner than they would have on an explicitly-safe processor. The new slack in the schedule can be exploited for higher throughput or lower power. With the VISA approach, an arbitrarily complex SMT processor can safely run non-real-time tasks at the same time as a real-time task. Alternatively, frequency/voltage can be safely lowered to take up slack. We explore the latter application and show a VISA-compliant complex pipeline consumes 43-61% less power than an explicitly-safe pipeline.

1. Introduction

In safe real-time systems (also known as hard real-time systems), correct operation depends on meeting deadlines [19]. Static worst-case timing analysis is vital to ensuring safe operation. The output of worst-case timing analysis is an upper bound on the execution time of a task, called the *worst-case execution time* (WCET). This bound is guaranteed never to be exceeded (safe bound) but is as close as possible to typical execution times (tight bound). Having WCET estimates is crucial for designing safe real-time systems, because they enable the system designer to budget enough processing power to handle worst-case computational requirements and safely meet deadlines under any circumstance.

Sophisticated timing analyzers can calculate safe, tight WCET bounds for tasks executing on single-issue in-order pipelines with instruction and data caches [2,11,12,14,15,16,17,18,26,34,42]. However, the level of sophistication needed to safely and accurately analyze more complex architectures is formidable. Currently, there is no way to precisely specify microarchitectures with a full complement of high-performance techniques (complex dynamic branch predictors, caches, deep speculation, dynamic scheduling, and multiple instruction issue), let alone safely and accurately predict WCET of tasks with variable control flow and data flow on these highly dynamic substrates. In fact, so far the simple pipeline described above is a *complexity limit* and simplicity may be fundamental to the design of safe real-time systems [4,8,28,32], because of the need for analyzability.

The complexity limit excludes an entire class of high-performance microprocessors from safe real-time systems. This has long-term implications in terms of expanding the scope of embedded systems in the future. Excluding high-performance microprocessors perpetuates the performance gap between general-purpose and embedded systems. As we will show, high-performance microprocessors can be exploited to increase throughput of multiprogrammed workloads and/or reduce power in safe real-time systems.

This paper presents a novel approach for reconciling the complexity/safety trade-off. We propose the notion of a *virtual simple architecture* (VISA) to decouple worst-case timing analysis from the underlying processor implementation. A VISA is the pipeline timing specification for a hypothetical, simple processor architecture. This hypothetical pipeline is the basis for worst-case timing analysis. However, the actual pipeline can be arbitrarily complex as long as progress of a task is continuously monitored and

shown to be no slower than the slowest progress on the hypothetical pipeline, as bounded by WCET of the hypothetical pipeline. Continuous monitoring is achieved by dividing the task into multiple smaller sub-tasks. Sub-tasks provide a mechanism to gauge progress of the task as it executes on the complex pipeline. An artificial interim deadline (called a checkpoint) is set for each sub-task. The checkpoint of a sub-task is based on the latest allowable completion time of the sub-task on the hypothetical pipeline. If sub-tasks complete before their respective checkpoints (as monitored by a watchdog timer), then it appears as if the hypothetical pipeline was used throughout. So, the task executed safely on the complex pipeline.

In the unlikely event that a sub-task does not complete before its checkpoint, then the overall task deadline may be missed unless steps are taken to bound execution time of the unfinished sub-task and all remaining sub-tasks. To address this, the complex pipeline is reconfigured to operate in a simple mode that directly implements the VISA, e.g., out-of-order (OOO) execution is disabled, *gshare* is replaced with static prediction, etc. The unfinished sub-task and all remaining sub-tasks are executed in simple mode, bounding execution time and thereby guaranteeing the task finishes before its hard deadline.

Disabling complex hardware features has been proposed before, e.g., disabling the cache or pinning critical cache lines [4,9]. Our approach differs fundamentally in that the real-time task is attempted on the complex pipeline even though it is unsafe to do so, and the run-time system dynamically and continuously verifies that the complex pipeline's execution time is bounded by WCET of the hypothetical simple pipeline. The simple mode is rarely or never used. Other approaches disable complex features for the duration of the real-time task, failing to exploit the high-performance microarchitecture. Another difference is that most disabling techniques target specific components, whereas we define a virtual simple architecture as a whole.

The significance of our approach is that we eliminate the need to do explicit WCET analysis of the complex pipeline, by confirming dynamically that its execution time is bounded by WCET of a hypothetical simple pipeline. This approach is analogous to the DIVA verification paradigm [3], in which proving correctness of a complex microarchitecture is avoided by dynamically confirming equivalence between the complex microarchitecture and a verified checker pipeline.

1.1. Exploiting unsafe processors in safe systems

In our experiments, tasks complete much sooner on the complex pipeline than on the hypothetical simple pipeline. In terms of safety, finishing earlier provides no benefit. However, the slack created by higher instruction-level parallelism (ILP) can be exploited to improve throughput of a multiprogrammed workload or reduce power consumption. We have identified at least three applications.

- *Conventional concurrency.* Typically, there is a mix of non-real-time, soft real-time, and hard real-time tasks in an embedded system. Finishing the hard real-time task earlier means non-real-time and soft real-time tasks can be scheduled during the slack following the hard real-time task.
- *Simultaneous multithreading (SMT).* In an SMT processor [37,41], slack can be exploited by executing non-real-time and soft real-time tasks at the same time as the hard real-time task. The hard real-time task only needs as much bandwidth as the hypothetical simple pipeline to meet sub-task checkpoints. On a highly-parallel microarchitecture, checkpoints are unlikely to be compromised by other tasks. However, if a checkpoint is missed, the complex pipeline is reconfigured to operate in simple mode, which now includes "idling" the other SMT threads (they are not context-switched out, but no new instructions are fetched).
- *Dynamic voltage scaling (DVS).* In a processor with DVS support, slack can be exploited by scaling down frequency and voltage, reducing power. Because it has higher ILP, the complex pipeline can meet its checkpoints at a lower frequency than the hypothetical simple pipeline. If a checkpoint is missed, the complex pipeline is reconfigured to operate in simple mode at a higher, safe frequency.

In this paper, we focus on the power savings application (dynamic voltage scaling). Experiments with 6 hard real-time benchmarks show that a VISA-compliant complex pipeline consumes 43% to 61% less power than an explicitly-safe pipeline.

The SMT application has major potential. Embedded systems have many threads, especially as dedicated hardware components are re-implemented in software threads (a trend called hardware-to-software migration) [7]. This environment is ripe for SMT processors. SMT can achieve significantly higher throughput than conventional concurrency. On the other hand, SMT poses new challenges for worst-case timing analysis. We may not know beforehand which threads will execute at the same time as the hard real-time task. And bounding WCET while other threads are executing — each with their own flow of control — is intractable. Our VISA approach would enable SMT processors to be used for higher throughput without compromising safety of critical tasks. The SMT application is beyond the scope of this paper and we leave it for future work.

1.2. Broader implications of VISA

The VISA is like another abstraction, the instruction-set architecture (ISA). In addition to providing a simple interface between software and hardware, the ISA provides a means for interoperability among different processors. Processors that conform to the same ISA are binary compatible, i.e., they correctly execute the same binaries. With VISA, the notion of binary compatibility could be extended to include timing safety. We could design different processors that conform to the same VISA. Parameterized WCET information for a task would be appended to the task's binary, and the task will execute safely within any system that complies with the VISA for which the WCET information was calculated (WCET would be expressed in cycles for frequency scaling, divided into components that scale and do not scale with frequency, and parameterized in terms of worst-case memory latency since the memory sub-system is outside the influence of processor design). The VISA could be formally appended to the ISA manual, with compliance optional. The broader implications of a VISA are left for future research.

2. General methodology for safe operation

In this section, we describe a general methodology for guaranteeing safe operation when using an unsafe, high-performance microarchitecture.

2.1. Setting checkpoints

To gauge progress on the unsafe pipeline, the task is divided into sub-tasks (by the programmer or timing analyzer) and each sub-task is assigned a soft deadline called a checkpoint. The sub-task is expected to complete before its checkpoint. If it does not, then the complex pipeline is re-configured to operate in a simple mode that directly implements the VISA. This way, the combined execution time of the unfinished sub-task and all remaining sub-tasks is safely bounded by worst-case timing analysis.

Checkpoints are the key to safe operation. For a sub-task *i* that misses its checkpoint, there must be just enough time between the checkpoint and the final deadline to (1) re-configure the complex pipeline to operate in simple mode (and switch frequency/voltage in the case of DVS), (2) execute the remainder of sub-task *i*, and (3) execute all remaining sub-tasks. Item 1 is a fixed implementation-dependent overhead. Item 3 is safely bounded by totaling the WCETs of remaining sub-tasks. Regarding item 2, it is hard to know how much time it will take to finish the remainder of sub-task *i*, because worst-case timing analysis is done for the sub-task as a whole. Fortunately, the remaining execution time is safely (although not tightly) bounded by WCET of the whole sub-task. That is, to be safe, we have to assume no work got done for sub-task *i* on the complex pipeline. Therefore, the checkpoint for sub-task *i* (relative to the beginning of the task) is calculated as follows:

$$checkpoint_i = \left(deadline - ovhd - \sum_{k=i}^{s} WCET_{k,f} \right) \quad \text{(EQ 1)}$$

Equation 1 corresponds to what was stated earlier, that the time between the checkpoint ($checkpoint_i$) and the final deadline (*deadline*) includes switching overhead (*ovhd*) plus the maximum time to execute the unfinished sub-task *i* and all remaining sub-tasks on the hypothetical simple pipeline. There are *s* sub-tasks in the task, so the WCETs of sub-tasks *i* through *s* are summed. For processors that support frequency/voltage scaling, there is a different WCET for each frequency setting. $WCET_{k,f}$ is the worst-case execution time for sub-task *k* at frequency *f* on the hypothetical simple pipeline.

2.2. Watchdog counter

A hardware cycle counter, called the watchdog counter, is used to detect missed checkpoints. The watchdog counter is memory-mapped so that it can be read and written via load and store instructions, respectively.

A code snippet at the beginning of the first sub-task initializes the watchdog counter to the number of cycles between the start of the task and the first checkpoint. Thus, the initial value of the watchdog counter is $\lfloor checkpoint_1 \cdot f \rfloor$, where *f* is the processor frequency. A code snippet at the beginning of each new sub-task *i* increments the watchdog counter by the number of cycles between the previous checkpoint ($checkpoint_{i-1}$) and the next one ($checkpoint_i$). Thus, sub-task *i* adds $\lfloor (checkpoint_i - checkpoint_{i-1}) \cdot f \rfloor$ cycles to the watchdog counter. This effectively advances the interim deadline enforced by the watchdog counter to the next checkpoint.

Meanwhile, hardware autonomously decrements the watchdog counter by one every cycle. If the watchdog counter reaches zero, it means the current sub-task missed its checkpoint. In this case, an exception is raised indicating that a checkpoint was missed.

Missed-checkpoint exceptions are masked if the processor is not running a hard real-time task, or if it is running a hard real-time task in simple mode. If a missed-checkpoint exception occurs and the processor is running a hard real-time task on the complex pipeline, then the pipeline is drained and re-configured to operate in simple mode.

3. System design

System design involves defining a VISA, and implementing a static timing analyzer on the one hand and a

complex processor on the other that comply with the VISA. In defining the VISA, we considered the capabilities of current timing analysis tools. We also considered how a simple mode of operation is likely to be accommodated within a typical dynamically scheduled superscalar processor, and in particular the complex processor used in this paper. The three layers — VISA, processor, and timing analyzer — are described in this section.

3.1. Virtual simple architecture

The VISA used in this paper is a six-stage, scalar (peak throughput is 1 instruction/cycle in all pipeline stages), in-order pipeline. The six pipeline stages are fetch, decode, register read, execute, memory, and writeback. The instruction fetch stage can fetch 1 instruction in a cycle. There is an instruction cache but no dynamic branch predictor. Conditional branches are predicted using a static heuristic: backward branches are predicted taken and forward branches are predicted not-taken. Branch target addresses are assumed to be cached with the branches in the instruction cache, i.e., the instruction cache and branch target buffer are merged. This simplifies static worst-case timing analysis since only the instruction cache needs to be analyzed. Targets of indirect branches are not predicted. Instruction fetch stalls until the indirect branch executes. There are four stages between fetch and execute. Therefore, the conditional branch misprediction penalty and indirect branch stall time are both four cycles.

The instruction decode stage decodes 1 instruction per cycle. There is no register renaming, because WAW hazards are handled by in-order completion and WAR hazards are handled by early-reads/late-writes. The register read stage checks for RAW hazards and reads source operands from the register file. There is a single, unpipelined, universal function unit in the execute stage, and all the usual data bypasses to the function unit from later pipeline stages. The universal function unit handles address generation and all integer and floating-point ALU instructions. There are only two causes for stalling an instruction in the register read stage. First, an instruction stalls in the register read stage if other instructions ahead of it in the pipeline are not advancing, due to a multiple-cycle operation in the execute stage or a data cache miss in the memory stage. Second, an instruction stalls in the register read stage if it depends on a load and the load is directly ahead of it in the pipeline (it must stall at least one cycle).

The VISA must also include specific cache configuration information, instruction execution latencies, and the worst-case memory stall time in the case of an instruction or data cache miss. The parameters used in this paper are shown in Table 1. Worst-case memory stall time is given in nanoseconds instead of cycles, because the number of cycles depends on the processor frequency.

TABLE 1. VISA caches and latencies.

component	parameters
L1 I-cache & D-cache	64KB, 4-way set-assoc., 64B block, 1 cycle hit
worst-case memory stall time	100 ns
execution latencies	MIPS R10K latencies

3.2. Pipeline alterations for simple mode

The complex processor used in this paper is a dynamically scheduled 4-way superscalar processor with a 128-entry reorder buffer, 64-entry issue queue, 64-entry load/store queue, 4 pipelined universal function units, and 2 ports to both the load/store queue and data cache. The pipeline has seven stages. These are fetch, dispatch, issue, register read, execute/memory, writeback, and retire. The caches and execution latencies are the same as for the VISA, shown in Table 1. The memory stall time can be worse than the stall time indicated in Table 1, due to contention in the memory system among multiple outstanding memory requests. However, without contention, the worst-case memory stall time is the same as for the VISA. A 2^{16}-entry *gshare* predictor is used to predict conditional branches [21]. A separate 2^{16}-entry table indexed the same way as the *gshare* predictor is used to predict indirect branch targets.

Simple mode uses the existing datapath of the complex processor. The following alterations are needed.

- The fetch unit disables the *gshare* conditional branch predictor and indirect target predictor. Instead, as specified in the VISA, forward conditional branches are predicted not-taken, backward conditional branches are predicted taken, and the fetch unit stalls until indirect branches execute.
- The fetch unit still retrieves a full fetch block from the instruction cache, as before, but the fetch block is buffered and its constituent instructions are passed down the pipeline at a maximum rate of one per cycle to comply with the VISA.
- A pipeline stage does not accept a new instruction from the previous stage if it already has an instruction and the instruction is not advancing in the next cycle.
- Renaming is still performed, but only to locate source and destination operands in the physical register file. Logical-to-physical mappings are never changed. Therefore, a new physical register does not need to be popped from the freelist and assigned to the destination operand, the rename map table does not need to be updated with a new mapping, and the rename map table does not need to be checkpointed at branches. In addition to disabling these three functions, the destina-

tion operand needs to be renamed like a source operand to locate it in the physical register file. Fortunately, a read port already exists for the destination operand, since the complex pipeline reads out previous mappings of destination operands and saves them in the active list for freelist maintenance. In simple mode, the previous mapping is used to rename the destination operand.

- An instruction in the dispatch stage advances directly to the register read stage, bypassing the issue queue. This removes the issue stage, conforming to the VISA pipeline stages. The instruction in the register read stage stalls if another instruction is tying up the execution stage or if it depends on an immediately-preceding load instruction, as specified in the VISA.
- Only one function unit is active at a time. An active function unit does not accept new instructions while in use, i.e., pipelining within function units is disabled. Note that the complex processor does not require a universal function unit as specified in the VISA, since any of its function units can be used just as long as only one is used at a time.
- Loads and stores do not access the load/store queue. Memory disambiguation is not needed because loads and stores access the data cache in program order. Stores issue to the data cache in the memory stage instead of waiting until commit, since all prior branches have already been resolved and stores are guaranteed to be non-speculative by the time they reach the memory stage.
- Instructions are not placed in the active list in the dispatch stage and they are removed from the pipeline after the writeback stage, bypassing retirement.

Some of these alterations are not strictly required for VISA-compliance. Specifically, the dispatch and retire stages (renaming, checkpointing, active list and freelist management, etc.) do not need alterations to get the same timing as the VISA. We did so in order to be explicit. It may also conserve power in simple mode.

The bypasses, bypass muxes, and bypass control logic do not need modifications. Complex mode and simple mode use the same hazard checking logic to detect when a value needs to be bypassed from the execute, memory, or writeback stage, to a dependent instruction as it begins execution.

Earlier we said that memory stall time may exceed the VISA worst-case memory stall time, due to contention among multiple outstanding memory requests. However, in simple mode, a load or store that misses in the cache stalls in the memory stage, ensuring there is only one outstanding memory request. So, worst-case memory stall time conforms to the VISA.

3.3. Static worst-case timing analysis

Dynamic timing analysis based on experimental or trace-driven approaches cannot be guaranteed to yield safe bounds of the worst-case execution time (WCET), mainly for two reasons. First, it is difficult to determine the worst-case input of real-time tasks with moderately complex input spaces. In most cases, exhaustive testing over the entire input space is simply not feasible. Second, even if the worst-case input with respect to the algorithmic properties of a program is known, hardware complexities, such as caching and pipelining, may cause the application to exhibit its worst-case behavior for a different input.

An alternative to dynamic timing analysis is given by static timing analysis. Static timing analysis provides the means to derive *safe* WCET estimates, i.e., the estimates provide a guaranteed upper bound on the computation time of a task. These bounds are a fundamental prerequisite for ensuring temporal correctness of applications according to schedulability tests, such as rate-monotone and earliest-deadline-first scheduling [19].

Static timing analysis performs the equivalent of a traversal over all execution paths to determine timing information independent of a program trace and without tracking values or program variables. Most significantly, loop bodies only require a few traversals to bound the WCET for the entire loop. We capture the worst-case behavior of architectural components along execution paths and compose these paths for loops, functions, and, ultimately, the entire application, to derive cycle counts that bound the WCET [1,2,11,23,24,25,26,27,38,39,40].

Figure 1 shows the organization of the timing analysis environment, which has been adapted to model the VISA and the Simplescalar instruction set (PISA) [6]. The application is compiled to assembly code using the gcc PISA compiler. Control flow and instruction/data memory references are extracted from the assembly code. In addition, upper bounds on the number of iterations for loops are provided.

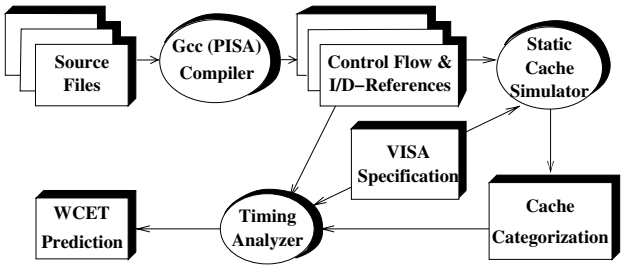

FIGURE 1. Static timing analysis toolset.

A static cache simulator uses the control flow information to construct a control-flow graph of the program that consists of the call graph and the control-flow graph of each function [2,23]. The program's control-flow graph is

then analyzed, and caching categorizations are derived for each instruction and data reference in the program [23]. Caching categorizations are described in Table 2. A separate categorization is given for each loop level in which the instruction and data references are contained, which significantly improves the derived bounds for loop nests. Categorizations are derived from the *abstract cache state*, which describes possible cache states before and after the execution of a basic block. Abstract cache states are calculated through iterative data-flow analysis. At joins in the control flow, abstract cache states are composed (as set unions), thereby providing may-analysis, which yields information about memory blocks that *may* be cached at execution points. In this manner, exponential overhead of modeling all possible cache states is avoided, and the approach has been shown to yield sufficiently tight worst-case bounds [26].

TABLE 2. Categorization of memory references.

always miss (m)	Reference not guaranteed to be in cache when accessed
always hit (h)	Reference guaranteed to be in cache when accessed
first miss (fm)	Reference not guaranteed to be in cache on its first reference for each loop execution, but guaranteed to be in cache on subsequent accesses
first hit (fh)	Reference guaranteed to be in cache on its first access for each loop execution, but not guaranteed to be in cache on subsequent accesses

Our tool has separate modules for static I-cache and D-cache analysis. The D-cache module has not been continuously maintained like the I-cache module. Our priority in this paper is modifying the timing analyzer (described next) with respect to the VISA specification and PISA instruction set. As such, for the time being, data cache misses are modeled by manually padding WCET based on data cache miss information from the dynamic trace. Future work includes re-integrating the D-cache module into the modified static timing framework.

The next component, the timing analyzer, uses the control flow information and loop bounds, caching categorizations, and pipeline description (the VISA) to derive timing predictions [2,11,12,39,40]. The pipeline simulator considers the effect of structural hazards (an instruction occupying the universal function unit for multiple cycles), data hazards (a load-dependent instruction stalls for at least one cycle if it immediately follows the load), branch prediction (backward-taken/forward-not-taken), and cache misses (derived from caching categorizations) for alternative execution paths through a loop body or a function. Static branch prediction is easily accommodated by worst-case analysis: the misprediction penalty is added to the non-predicted path (not-taken path for backward branches and taken path for forward branches) and path analysis (below) selects the longest path as usual.

Once timings for alternate paths in a loop are obtained, a fix-point algorithm (iterative solution that converges) is employed to safely bound the time of the loop based on the cycle counts of the loop body. Typically, the approach requires path analysis for only a few iterations. Given the longest path for the first iteration, the next-longest path is determined for the second iteration, which may differ from the original path due to caching effects. The lengths of these paths are monotonically decreasing due to cache effects, and once we reach a fix-point, subsequent loop iterations can be safely approximated by this fix-point timing value. Note, when the longest paths of consecutive iterations are combined, we account for the pipeline overlap between the tail of the earlier path and the head of the path that follows (the alternative - no overlap - is tantamount to draining the pipeline between iterations).

Using this fix-point approach, the timing analyzer then derives WCET bounds, first for each path, then for loops, and finally for functions within the program. A timing analysis tree is constructed, where each node of the tree corresponds to a loop or function. Nodes in the tree are processed in a bottom-up manner. In other words, the WCET for an outer loop / function is not calculated until the times for all of its inner loops / called functions are known. This means that the timing analyzer predicts the WCET for programs by first analyzing the innermost loops and functions before proceeding to higher level loops and functions until it reaches the top level (e.g., main()). For our purposes, the timing analysis tree provides a convenient method for obtaining WCET for a specific scope, and sub-tasks in particular.

From the description in this section, it becomes more evident that static timing analysis is non-trivial even for simple pipelines. This provides further evidence of the need for a VISA to build safe real-time systems from complex components.

4. Exploiting slack for power savings

To exploit slack for power savings, we adapt the frequency speculation technique developed by Rotenberg [35] to our framework. Frequency speculation was initially proposed as a way to reconcile the potentially large gap between worst-case execution time and typical execution time. Rotenberg conjectured that it may be possible to do safe worst-case timing analysis of complex pipelines in the future, but that the bounds would not be tight. The result is that the processor runs at a much higher frequency than needed. Frequency speculation addresses this problem, as follows. A task is divided into multiple sub-tasks. Off-line simulation is used to find a tight but unsafe bound on the number of cycles for a sub-task. This bound is the basis for a low speculative frequency. As long as sub-tasks do not exceed their bounds, the overall deadline is met in spite of running at an unsafe frequency. If a sub-task exceeds its

bound (called a misprediction), then the processor switches to a higher recovery frequency. The recovery frequency is based on a safe bound on the number of cycles for remaining sub-tasks, thereby guaranteeing the overall deadline is met in spite of the interim misprediction.

Frequency speculation successfully reduces frequency. However, a crucial limitation is that safe worst-case timing analysis of the complex microarchitecture is still required to guarantee the deadline [35]. Yet, this may not be possible. The VISA abstraction eliminates this requirement. We begin by reviewing conventional frequency speculation, and then adapt it to the VISA framework.

4.1. Conventional frequency speculation

Frequency speculation uses two sets of execution times for each sub-task: worst-case execution times (WCET) and predicted execution times (PET) at all frequencies. WCET is computed by static worst-case timing analysis and PET is based on measurements (e.g., off-line simulation [35]). WCET is always greater than PET for a given frequency. Frequency speculation, as originally proposed, is summarized by the following expression.

$$\sum_{j=1}^{i-1} PET_{j,f_{spec}} + WCET_{i,f_{spec}} + ovhd + \sum_{k=i+1}^{s} WCET_{k,f_{rec}} \leq deadline \quad \text{(EQ 2)}$$

The first term in Equation 2 indicates that sub-tasks 1 through *i*-1 are not mispredicted, because their combined execution time is bounded by the sum of their PETs at the speculative frequency. The second term indicates that sub-task *i* is mispredicted, because its execution time is bounded by WCET at the speculative frequency, not PET. At this point, continuing at the speculative frequency is unsafe. To guarantee the overall deadline, the processor switches to a higher recovery frequency and remaining sub-tasks are not speculated, i.e., WCETs are used instead of PETs to safely bound their combined execution time. The third term charges a fixed overhead to change the frequency/voltage and the fourth term safely bounds the execution time of remaining sub-tasks at the recovery frequency. The sum of the four terms must fit within the deadline. Equation 2 actually represents *s* different equations, since any one of the sub-tasks may be mispredicted. A simple iterative technique solves for the lowest $\{f_{spec}, f_{rec}\}$ pair that satisfies all *s* equations [35].

4.2. Adapting speculation to the VISA framework

We now adapt frequency speculation to our framework. A key benefit of our framework is that static worst-case timing analysis does not need to be done for the complex microarchitecture. Recovery involves switching to a safe recovery frequency and switching to the simple mode of operation. By also switching to the simple mode, the combined execution time of remaining sub-tasks is safely bounded without having to analyze the complex microarchitecture.

The second term in Equation 2 has to be modified. It assumes the execution time of the mispredicted sub-task *i* can be safely bounded on the complex processor. This is not the case in our framework. The watchdog counter indicates that sub-task *i* is mispredicted. Attempting to finish the mispredicted sub-task before switching to simple mode is unsafe, because we cannot bound how much longer it will take to finish it on the complex processor. Yet, if the processor is switched to simple mode as soon as the misprediction is detected, we can bound how much longer sub-task *i* will take to finish: it cannot take longer than its worst-case execution time on the VISA. Accordingly, the second term in Equation 2 is split into two terms to handle sub-task *i* before and after switching to simple mode. The two terms are highlighted in bold in Equation 3 below.

$$\sum_{j=1}^{i-1} PET_{j,f_{spec}} + \boldsymbol{PET_{i,f_{spec}}} + ovhd + \boldsymbol{WCET_{i,f_{rec}}} + \sum_{k=i+1}^{s} WCET_{k,f_{rec}} \leq deadline \quad \text{(EQ 3)}$$

The second, third, and fourth terms indicate that sub-task *i* is mispredicted. The second term is the predicted execution time of sub-task *i* at the speculative frequency. This is the amount of time that elapses before detecting that more time is needed to finish sub-task *i*. At that point, the watchdog counter raises an exception, the frequency is switched to the recovery frequency, and simple mode is initiated. The third term accounts for the time needed to switch frequency and mode. The fourth term safely bounds the execution time of the unfinished portion of sub-task *i* at the recovery frequency. The tightest bound we can safely use is the WCET for the entire sub-task *i*, because worst-case analysis is done for the sub-task as a whole (as explained earlier in Section 2.1). Equation 3 is simplified by merging the speculation and recovery terms for sub-task *i* into the corresponding sum terms, as shown in Equation 4 below.

$$\sum_{j=1}^{i} PET_{j,f_{spec}} + ovhd + \sum_{k=i}^{s} WCET_{k,f_{rec}} \leq deadline \quad \text{(EQ 4)}$$

The same iterative method developed for frequency speculation [35] is used to find the minimum $\{f_{spec}, f_{rec}\}$ pair. With these two frequencies, we can set the sub-task checkpoints and determine how many cycles each sub-task adds to the watchdog counter, using the procedures outlined in Section 2. Checkpoints are set as explained in

Section 2.1, using the recovery frequency f_{rec} for the frequency f in Equation 1. Section 2.2 explains how to compute the number of cycles each sub-task adds to the watchdog counter. In this case, the speculative frequency f_{spec} is used for the frequency f.

4.3. Selecting predicted execution times

Previously, off-line simulation was used to select predicted execution times (PET) of sub-tasks [35]. We use run-time profiling in this paper.

Another memory-mapped cycle counter is provided by the processor to measure the number of cycles for sub-tasks. The counter increments by one every cycle. A code snippet at the beginning of a sub-task resets the cycle counter. A code snippet at the end of the sub-task queries the cycle counter to obtain the actual execution time (AET). Each sub-task records its own AET history.

We conceived of two ways to record AET history. In the first approach, AETs are recorded in a histogram. In the second approach, AETs of only the last N instances of a sub-task are recorded.

AET of a mispredicted sub-task cannot be recorded precisely because the unfinished portion is executed in simple mode, artificially inflating AET. A solution is to scale down the number of cycles spent in simple mode by some factor, based on the relative performance of the complex and simple modes.

PETs for sub-tasks are re-evaluated periodically based on AET histories. In this paper, we re-evaluate PETs every tenth time a task is executed. For the last-N approach, PET is set to the maximum of the last 10 AETs. The histogram approach provides a probabilistic method for targeting a certain misprediction rate. For example, if we want to target zero mispredictions for a particular sub-task, PET is set such that 0% of the recorded AETs are higher. If we want to target a 10% probability of mispredicting, PET is set such that 10% of the recorded AETs are higher. Targeting a non-zero misprediction rate may result in a lower speculative frequency. However, this must be weighed against running in high-power recovery mode more often.

After PETs are re-evaluated, the speculative and recovery frequencies for the task are re-computed using the procedure outlined in Section 4.2. Likewise, new checkpoints are set and watchdog increment values are re-computed for each sub-task, as outlined in Section 2.

5. Experimental method

5.1. Cycle-accurate simulator

A detailed cycle-accurate simulator models the complex processor, including its simple mode of operation. Processor parameters were given in Section 3.2. The Simplescalar ISA (PISA) [6] is used. Memory-mapped counters (watchdog counter, cycle counter) and registers (current frequency register, recovery frequency register) are modeled in the processor simulator.

5.2. Power modeling

We integrated the Wattch power models [5] into our simulator to measure power and energy. The models were modified to closely match the structures in a contemporary superscalar microarchitecture (separate physical register file, active list, issue queue, and load/store queue) instead of the default RUU-based microarchitecture. Support was also added for dynamic voltage scaling. The frequency/voltage settings used for DVS are loosely based on Intel Xscale, which is reported to have 5 settings ranging from 150 MHz / 0.76 V to 1 GHz / 1.8 V [43]. From the Xscale, we extrapolated 37 settings ranging from 100 MHz / 0.70 V to 1 GHz / 1.8 V in 25 MHz / 0.03 V increments.

Wattch power numbers are reported using perfect (proportional) clock gating, with and without 10% standby power [5]. In Wattch, standby power accounts for power that may still be consumed in an otherwise completely idle unit.

The base processor used for power comparisons is a literal implementation of the VISA. The base case is called *simple-fixed* to distinguish it from the simple mode of operation in the complex processor. For the same frequency and voltage, *simple-fixed* is more power-efficient than simple mode because its structures are sized exactly according to the VISA and it has no extraneous structures. For example, the integer register file of *simple-fixed* contains only 32 registers. On the other hand, simple mode accesses a large physical register file even though only 32 integer registers are actually used, and a limited form of renaming is still needed to locate the registers in the physical register file.

In Wattch, dimensions of the die affect a few of the power models (e.g., die length is used to estimate the lengths of global clock wires). Although the caches consume a lot of die area in both the complex and *simple-fixed* processors, we halved both die dimensions for *simple-fixed* since it has fewer structures and a scalar pipeline.

Another aspect we consider is the possible frequency advantage of *simple-fixed*. For the same voltage, it might be possible to clock *simple-fixed* faster than the complex processor. This aspect is difficult to ascertain. If the caches are as critical as the rename logic, wakeup/select logic, superscalar bypasses, etc., then *simple-fixed* does not have a frequency advantage over the complex processor. In any case, it is beyond the scope of this paper to quantify cycle time differences. But, to model the potential frequency advantage, we do some experiments in which *simple-fixed* has 1.5 times the frequency of the complex processor for a given voltage.

Power overhead of code snippets added before and after sub-tasks is accounted for. These snippets advance the watchdog counter and maintain the execution time histories. Also included in the power measurement is the DVS software that executes every tenth task. This software re-evaluates PETs based on the execution time histories (the last-N approach is used in all experiments), re-computes the speculative and recovery frequencies, and re-computes checkpoints and watchdog increments.

For both the complex processor and *simple-fixed*, if a task completes before its deadline, the frequency is lowered to 100 MHz (lowest setting) to conserve power until the deadline is reached.

5.3. Benchmarks and sub-task selection

We use six different benchmarks from the C-lab real-time benchmark suite [44], shown in Table 3. The C-lab benchmarks are used extensively in WCET research, in particular because irregular program features that foil static timing analysis are explicitly avoided, which is typical of hard real-time code. The benchmarks are compiled with -O3 optimization enabled.

TABLE 3. C-lab benchmarks.

		adpcm	cnt	fft	lms	mm	srt
# dyn. inst. 1 task		1.99M	70.6K	219K	113K	1.65M	1.63M
tight dead. (ms)		3.7	0.090	0.50	0.19	2.3	3.9
loose dead. (ms)		5.2	0.12	0.66	0.29	3.4	5.8
# of sub-tasks		8	5	10	10	10	10
time @ 1GHz for 1 task (µs)	WCET	3,286	72	426	173	2,056	3,508
	actual time: simple	2,428	71	368	168	2,050	1,755
	actual time: complex	648	21	63	40	658	506
time ratios	WCET/ simple	1.35	1.01	1.16	1.03	1.00	2.00
	simple/ complex	3.75	3.38	5.84	4.23	3.12	3.47

Two deadlines are used for each benchmark, one tight and one loose, shown in Table 3. The tight deadline is the tightest that can be guaranteed with frequency speculation (which introduces overheads), hence it typically yields frequencies over 800MHz for *simple-fixed*. The basis for the loose deadline is an intermediate frequency of around 600MHz for *simple-fixed*.

Sub-task selection is done manually. Typically, a sufficient number of balanced sub-tasks can be achieved by peeling off chunks of iterations from the outermost loop. Code segments before and after the outermost loop are merged into the first and last sub-tasks, respectively. Table 3 shows the number of sub-tasks in each benchmark.

In each experiment, a task is executed 200 consecutive times as if to model a periodic real-time task.

6. Results

6.1. Static timing analysis results

Table 3 shows various execution times for a single task at 1 GHz. The "WCET" row is the worst-case execution time bound calculated by the timing analyzer. The "simple" and "complex" rows are actual execution times on the *simple-fixed* and complex processors, respectively. The bounds from the timing analyzer are close to actual execution times on the *simple-fixed* processor for all but *srt* and *adpcm*. Excluding *srt* and *adpcm*, the timing analyzer over-estimates execution time by at most a factor of 1.16 (Table 3, row labeled "WCET/simple").

The *srt* task is an implementation of the bubblesort algorithm. There are two sources of over-estimation. First, there are many forward branches that test whether elements should be swapped or not. Worst-case analysis always assumes the longest control-dependent path. Second, the sub-tasks get progressively smaller as the array gradually becomes sorted. This is manifested as an early exit from the loop kernel. Worst-case analysis assumes the loop is not exited early.

As expected, the complex processor is much faster than *simple-fixed*, by a factor of about 3 to 6 (Table 3, row labeled "simple/complex"). The slack can be exploited for low power, while VISA-compliance ensures safety.

6.2. Safe operation with low power

We now compare the power of the VISA-compliant complex processor and the explicitly-safe processor (*simple-fixed*). Note that *simple-fixed* can also benefit from frequency speculation, in particular for the three benchmarks for which WCET is inflated (*adpcm*, *fft*, *srt*). On the other hand, for the three benchmarks for which WCET is tight (*cnt*, *lms*, *mm*), frequency speculation actually increases frequency with respect to not using frequency speculation because misprediction overhead must be budgeted. Therefore, frequency speculation is only used by the *simple-fixed* processor when it reduces frequency.

Figure 2 shows the power savings of the complex processor relative to *simple-fixed*, for both tight (T) and loose (L) deadlines. As expected, the complex processor saves power, ranging from 43-61% less power than *simple-fixed* for tight deadlines and no standby power. (Savings are even higher with 10% standby power.) This is due to the fact that *simple-fixed* runs between 800 and 900 MHz whereas the complex processor runs between 150 and 325 MHz, depending on the benchmark. Also, the complex processor spends no time in simple mode because there is little variation in execution time which means PETs are accurate and no checkpoints are missed. Power savings are less for the loose deadline than the tight deadline, but still substantial, ranging from 22-48% without standby power.

As deadlines are loosened, power decreases more rapidly for *simple-fixed* than the complex processor, because *simple-fixed* requires a much higher frequency to begin with. For the loose deadline, *simple-fixed* runs between 375 and 600 MHz and the complex processor runs between 125 and 225 MHz, depending on the benchmark.

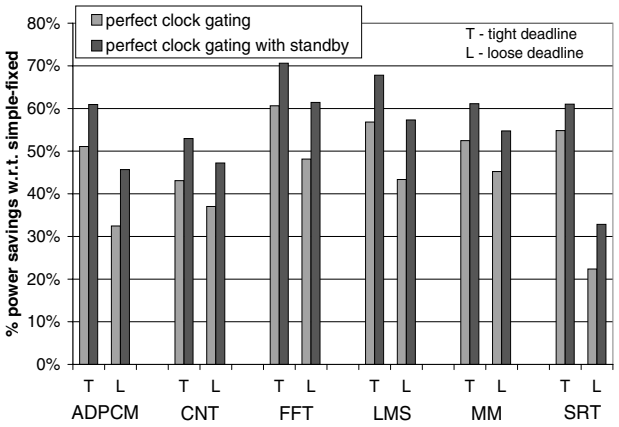

FIGURE 2. Power savings of the VISA-compliant complex processor relative to *simple-fixed*.

Figure 3 shows results for the case in which *simple-fixed* can run at 1.5 times the frequency of the complex processor for a given voltage. The tight deadline is used. As expected, power savings are less than before, but still substantial. The complex processor consumes 10-38% less power than *simple-fixed* (without standby power).

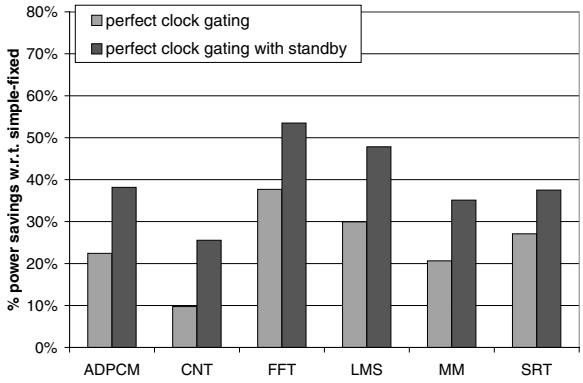

FIGURE 3. Comparison in which *simple-fixed* processor has 1.5x frequency.

Finally, results with 10%, 20%, and 30% of the tasks mispredicted are shown in Figure 4. The tight deadline is used. To induce missed checkpoints, we flushed the caches and branch predictor at the beginning of 20, 40, or 60 of the tasks (out of 200 tasks total). For the same deadlines used previously, checkpoints were missed in only three of the benchmarks using this method (*cnt*, *lms*, *srt*). (There was residual slack in the other three benchmarks.) As seen, the decline in power savings is proportional to the misprediction rate, since the complex processor executes mispredicted tasks almost entirely in simple mode. Note that even though mispredictions occur, all deadlines are safely met.

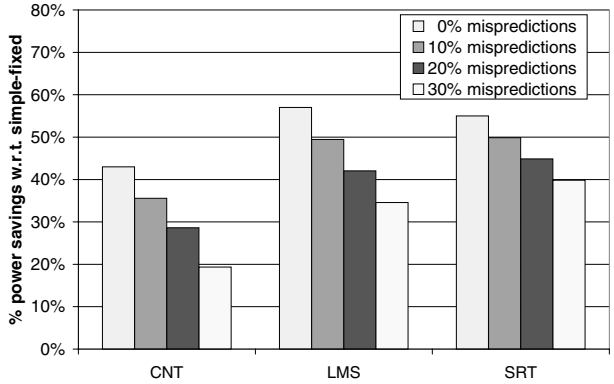

FIGURE 4. Power savings of the VISA-compliant complex processor with mispredicted tasks.

7. Related work

Simplicity is often cited as a prerequisite for safe real-time systems [e.g.,4,8,28,32]. It has also been noted that complex architectural features are allowable as long as they are disabled based on analyzability requirements [4,9]. While our VISA approach also involves disabling complexity, this aspect is rarely used because the complex pipeline is usually safe although it cannot be statically proven. Timeliness is verified dynamically which allows critical tasks to run safely on what are considered to be unsafe components. As pointed out earlier, the dynamic verification aspect is related to the DIVA verification paradigm [3], except that we dynamically verify timeliness instead of functionality. As such, the simple mode of operation is a subset of the complex pipeline rather than a dedicated processor.

Over the past decade, various research groups have investigated static approaches for bounding WCET of real-time programs. Static analysis has been extended from unoptimized programs on simple CISC processors [10,29,30,33] to optimized programs on pipelined RISC processors [12,18,42], and from uncached architectures to architectures with instruction caches [2,14,16,26] and data caches [11,15,17,34]. Lundqvist and Stenstrom modified an architectural simulator to determine WCET bounds by considering alternate execution paths in parallel (instead of following a trace) combined with pruning techniques to reduce the search space [20]. We fully leverage all of the above work with respect to deriving WCET for tasks on the hypothetical simple pipeline. While we expect continued success in extending timing analysis to increasingly complex processors, this effort will always lag behind microprocessor technology. The VISA framework may expedite the use of complex processors in safe systems, at the same time fully leveraging the significant investment in timing analysis tools.

Recently, a method has been proposed to bound the number of branch mispredictions for two-level dynamic predictors [22]. Even if analyzing complex pipelines remains intractable, techniques for bounding mispredictions are very useful in the context of a VISA.

Hughes, Srinivasan, and Adve [13] explore the energy-savings potential of architectural adaptations and DVS, separately and in combination. One of their key results is that complex microarchitectures are often more energy-efficient than simpler microarchitectures in the context of DVS, because the same performance can be achieved with lower frequency. Their work is a motivating factor for exploiting slack for power savings in this paper.

We borrowed and adapted the frequency speculation technique proposed by Rotenberg [35] to achieve power savings. A key limitation of that work is that static worst-case timing analysis is still required for the complex processor. Our VISA framework is an ideal match to alleviate this requirement. With our simple adaptation of the algorithm, worst-case analysis of the complex processor is no longer needed.

VLIW and delay slots are other examples of architecting pipeline timing.

8. Summary and future work

Worst-case timing analysis is a key component in safe real-time systems. Due to analyzability requirements, complex processors are either excluded altogether or their complex features are disabled during the execution of hard real-time tasks. This has implications in terms of expanding the scope of embedded systems in the future.

We proposed a novel approach for reconciling the complexity/safety trade-off that leverages the significant investment in static worst-case timing analysis, but, at the same time, represents a departure from this direct approach to guaranteeing safety. Namely, worst-case timing analysis is decoupled from the underlying processor implementation through a virtual simple architecture (VISA). The VISA is the basis for worst-case timing analysis but the underlying implementation can be arbitrarily complex. The key innovation is dynamically and continuously monitoring progress of a task on the unsafe pipeline, thereby confirming that the unsafe pipeline is as timely as the hypothetical simple pipeline (the VISA), as bounded by worst-case timing analysis of the VISA proxy. Subtasks provide a mechanism for gauging progress. If subtasks meet their checkpoints (interim deadlines based on latest allowable completion time on the hypothetical simple pipeline), then overall task safety is ensured in spite of executing on an unsafe pipeline. If any sub-task misses its checkpoint, the complex pipeline switches to a simple mode of operation that directly implements the VISA, safely bounding execution time of remaining sub-tasks.

The VISA approach provides a general framework for safe operation on unsafe processors, and sets up various opportunities for exploiting higher performance. We showed that the high ILP of the complex processor can be leveraged to lower frequency and voltage, resulting in power savings of 43% to 61% compared to an explicitly-safe pipeline. In future work, we plan to leverage high ILP for higher throughput via simultaneous multithreading. The idea is to exploit newly-created slack in the schedule of the critical task by simultaneously executing other soft real-time and non-real-time tasks. The VISA framework ensures overall safety. Usually, the critical task will have sufficient resources to meet its checkpoints, otherwise the simple mode of operation idles non-critical tasks (relinquishing bandwidth without swapping them out of the processor). Embedded systems tend to have a lot of concurrency, so enabling the unrestricted use of arbitrarily complex SMT processors in safe real-time systems has major potential.

Crafting a VISA that is easily accommodated within a wide variety of microarchitectures is an area of future research. We will also investigate other methods for ensuring VISA-compliance that do not require a simple mode of operation, such as the notion of compliance-by-design. Finally, the VISA abstraction may have broader implications with further research. Parameterized WCET information could be appended to a task's binary, and the task will execute safely within any system that complies with the VISA for which WCET was calculated — extending the notion of binary compatibility to include timing safety.

9. Acknowledgments

This research was supported in part by NSF grants CCR-0207785 and CCR-0208581, NSF CAREER grant CCR-0092832, and generous funding and equipment donations from Intel.

10. References

[1] R. Arnold. Bounding instruction cache performance. M.S. Thesis, Dept. of CS, Florida State Univ., Dec. 1996.

[2] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. *Real-Time Systems Symp.*, Dec. 1994.

[3] T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. *32nd Int'l Symp. on Micro.*, Nov. 1999.

[4] I. Bate, P. Conmy, T. Kelly, and J. McDermid. Use of modern processors in safety critical applications. *The Computer Journal*, 44(6):531–543, 2001.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *27th Int'l Symp. on Comp. Arch.*, June 2000.

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

[7] A. Dean and J. P. Shen. Hardware to software migration with real-time thread integration. *EuroMicro Workshop on Digital System Design*, August 1998.

[8] J. Engblom. On hardware and hardware models for embedded real-time systems. *IEEE Workshop on Real-Time Embedded Systems*, Dec. 2001.

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

[10] M. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. *Real-Time Systems Symp.*, Dec. 1992.

[11] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. on Computers*, 48(1):53–70, Jan. 1999.

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

[13] C. J. Hughes, J. Srinivasan, and S. V. Adve. Saving Energy with Architectural and Frequency Adaptations for Multimedia Applications. *34th Int'l Symp. on Microarchitecture*, Dec. 2001.

[14] Y. Hur, Y. H. Bae, S.-S. Lim, B.-D. Rhee, S. L. Min, C. Y. Park, M. Lee, H. Shin, and C. S. Kim. Worst case timing analysis of RISC processors: R3000/R3010 case study. *Real-Time Systems Symposium*, Dec. 1995.

[15] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. *Real-Time Technology and Applications Symposium*, June 1996.

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

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

[18] 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. *Real-Time Sys. Symp.*, Dec. 1994.

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

[20] 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.

[21] S. McFarling. Combining branch predictors. Technical Report TN-36, WRL, June 1993.

[22] T. Mitra and A. Roychoudhury. A Framework to Model Branch Prediction for WCET Analysis. *2nd Workshop on Worst Case Execution Time Analysis (WCET)*, June 2002.

[23] F. Mueller. Static Cache Simulation and its Applications. Ph.D. Thesis, Dept. of CS, Florida State Univ., July 1994.

[24] F. Mueller. Generalizing timing predictions to set-associative caches. *EuroMicro Workshop on Real-Time Sys.*, June 1997.

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

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

[27] F. Mueller, D. B. Whalley, and M. Harmon. Predicting instruction cache behavior. *ACM Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[28] D. Niehaus, E. Nahum, J. Stankovic, and K.Ramamritham. Architecture and O/S support for predictable real-time systems. *Spring internal document*, March 1992.

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

[30] P. Puschner. Zeitanalyse von Echtzeitprogrammen. Ph.D. Thesis, Dept. of CS, Technical University Vienna, Dec. 1993.

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

[32] P. Puschner. Is worst-case execution-time analysis a non-problem? — towards new software and hardware architectures. *2nd EuroMicro Int'l Workshop on WCET Analysis*, June 2002.

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

[34] J. Rawat. Static analysis of cache analysis for real-time programming. M.S. Thesis, Iowa State Univ., 1995.

[35] E. Rotenberg. Using Variable-MHz Microprocessors to Efficiently Handle Uncertainty in Real-Time Systems. *34th International Symposium on Microarchitecture*, December 2001.

[36] H. Theiling and C. Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. *Real-Time Systems Symposium*, Dec. 1998.

[37] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *22nd Int'l Symp. on Computer Architecture*, June 1995.

[38] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, Nov. 2001.

[39] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. *Real-Time Technology and Applications Symposium*, June 1997.

[40] R. T. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, Nov. 1999.

[41] W. Yamamoto and M. Nemirovsky. Increasing Superscalar Performance through Multistreaming. *Parallel Architectures and Compilation Techniques*, June 1995.

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

[43] http://developer.intel.com/design/intelxscale/benchmarks.htm

[44] "C-lab: WCET Benchmarks," http://www.c-lab.de/home/en/download.html.