Using Variable-MHz Microprocessors to Efficiently Handle Uncertainty in Real-Time Systems

Eric Rotenberg
Center for Embedded Systems Research (CESR)
Department of Electrical and Computer Engineering
North Carolina State University

Abstract

Guaranteed performance is critical in real-time systems because correct operation requires tasks complete on time. Meanwhile, as software complexity increases and deadlines tighten, embedded processors inherit high-performance techniques such as pipelining, caches, and branch prediction. Guaranteeing the performance of complex pipelines is difficult and worst-case analysis often under-estimates the microarchitecture for correctness. Ultimately, the designer must turn to clock frequency as a reliable source of performance. The chosen processor has a higher frequency than is needed most of the time, to compensate for uncertain hardware enhancements — partly defeating their intended purpose.

We propose using microarchitecture simulation to produce accurate but not guaranteed-correct worst-case performance bounds. The primary clock frequency is chosen based on simulated-worst-case performance. Since static analysis cannot confirm simulated-worst-case bounds, the microarchitecture is also backed up by clock frequency reserves. When running a task, the processor periodically checks for interim microarchitecture performance failures. These are expected to be rare, but frequency reserves are available to guarantee the final deadline is met in spite of interim failures.

Experiments demonstrate significant frequency reductions, e.g., -100 MHz for a peak 300 MHz processor. The more conservative worst-case analysis is, the larger the frequency reduction. The shorter the deadline, the larger the frequency reduction. And reserve frequency is generally no worse than the high frequency produced by conventional worst-case analysis, i.e., the system degrades gracefully in the presence of transient performance faults.

1. Introduction

Performance does not affect correctness for ordinary general-purpose programs. On the other hand, a real-time program must complete within a specified period of time, i.e., performance does affect correctness in real-time systems. Therefore, an important criterion when selecting a microprocessor for a real-time system is *guaranteed performance*.

More tasks, more instructions per task, and tighter real-time deadlines due to evolving specifications increase the complexity of real-time systems and demand higher performance. As a result, pipelining, caching, branch prediction, and even out-of-order and multiple-instruction issue are finding their way into embedded microprocessors.

Unfortunately, the performance of complex pipelines is difficult to guarantee. For example, cache performance is uncertain due to statically-unknown load and store addresses. In general, the interaction between ambiguous program information and history-sensitive hardware introduces uncertainty. In real-time systems, uncertainty is handled by designing for worst-case behavior [9]. At one extreme, for example, the designer may have to assume a particular load instruction always misses in the data cache.

The paradox is that pipelining, caches, and predictors are added to enhance performance so that more aggressive deadlines can be met, but their combined performance is underestimated to guarantee correct operation. Ultimately, the system designer must resort to clock frequency as a predictable and reliable source of performance. Conservatism is tantamount to not fully exploiting microarchitectural performance and compensating with abundant clock frequency. So, *redundant performance* is built into the system, i.e., the design has both a high performance microarchitecture and a high clock frequency.

Redundant performance is certainly required if static analysis cannot confirm that the microarchitecture will perform reliably *all of the time*. But, over-compensating with clock frequency, which we call *over-design*, has two serious problems.

- It is inefficient to over-compensate with clock frequency all of the time — especially when the microarchitecture is expected to perform well most of the time, and in spite of not being able to guarantee it with absolute certainty. In practice, the predictor, caches, and pipeline are carefully selected to perform well most or all of the time, for a specific embedded application that is unlikely to change over the lifetime of the system.

- The guaranteed-correct worst-case bound predicted by static analysis may be highly exaggerated compared to worst-case performance that occurs in practice. In this case, the over-designed frequency is highly inflated.

This paper proposes a new way of hedging microarchitecture performance in real-time systems. Clock frequency is chosen based on accurate estimates of worst-case microarchitecture performance, produced by simulation. The estimated worst-case bounds are not provably correct. So, the microarchitecture is backed up by extra clock frequency *reserves* that are only used if the microarchitecture fails.

Missing a task deadline is the only way to know for certain that the microarchitecture failed, but that defeats our purpose. The next best thing is to periodically assess the *possibility* of missing the deadline. Mossé, Aydin, Childers, and Melhem [17] proposed dividing a task into multiple smaller sub-tasks, which introduces periodic points for managing the processor (in their case, *dynamic power management*). Using sub-tasks enables us to set up artificial interim deadlines, called checkpoints, that can be used to detect interim microarchitecture failures. An interim microarchitecture failure does not necessarily mean the final deadline will be missed. However, we conservatively assume that an interim failure will lead to an overall failure. *Transient performance faults* are expected to be rare, similar to transient hardware faults.

Thorough simulation is used to bound worst-case performance of the microarchitecture, and the primary frequency of the processor is based on this bound. Simulation may not produce the worst possible scenario, therefore, the primary frequency is speculative. The processor attempts sub-tasks at the *speculative frequency.* According to simulation, sub-tasks are expected to meet their checkpoints at the speculative frequency, but static analysis cannot confirm this. So, when a sub-task completes, the processor checks to see if the sub-task's checkpoint was missed. If it was, the processor resorts to its clock frequency reserves. Remaining sub-tasks are run at a higher *recovery frequency* that guarantees the final deadline is met, in spite of the interim microarchitecture failure. This paper develops a method for statically deriving the speculative and recovery frequencies.

1.1 Potential advantages

Although the new approach for hedging microarchitecture performance still requires high frequency support, using high frequency sparingly (or not at all) does have potential benefits. By favoring microarchitectural sources of performance (instruction-level parallelism) over clock frequency, power consumption may be less for the same deadline. Others have derived that running at a lower frequency for an extended period consumes less power than running full throttle for a short period and then idling, if both voltage and frequency are scaled [5].

Hedging the microarchitecture with frequency reserves may also relax the need for increasingly sophisticated *worst-case execution time* (WCET) analysis. High-performance microarchitectures pose difficult challenges for tight WCET analysis [e.g., 1,3,9,10,13,14,16]. Using accurate simulation to drive the design and hedging speculation with high frequency removes some of the burden from static WCET analysis. This reduces the burden on compiler developers, in the case of automated WCET analysis, and programmers, in the case of manual WCET analysis.

Relaxing the need for tighter WCET analysis also reduces the risk of bugs. Sophisticated WCET analysis is more susceptible to bugs than simple WCET analysis. Note that even our frequency speculation technique relies on WCET analysis, because the deadline must be guaranteed.

Finally, a simulation-based approach to real-time system design may promote programming styles that were once discouraged because they make WCET analysis more difficult. This in turn potentially increases programmer productivity and enables more complex real-time software.

1.2 Target microprocessors

The frequency speculation technique will work with general-purpose microprocessors that provide many distinct frequency/voltage settings, such as the Transmeta Crusoe processors. Most dynamic power management proposals target these flexible processors [e.g., 17].

Yet, because the speculative and recovery frequencies are customized to the embedded system application, a *custom-fit processor* [4] is a compelling alternative. There are many possibilities, some of which are described below.

- *Custom-fit processor with two frequency/voltage settings*. The processor supports two frequency/voltage settings: the speculative frequency with low voltage and the recovery frequency with high voltage. Designing and verifying a pipeline with only two settings may be much simpler than designing and verifying a pipeline with many settings, at the expense of flexibility.

- *Custom-fit processor with dual pipeline*. The primary pipeline is designed at the speculative frequency and low voltage. A backup pipeline is designed at the recovery frequency and high voltage. The recovery pipeline is switched on as needed. The advantage of this approach is that each pipeline is designed to operate at only one frequency/voltage setting, simplifying design and verification. Another advantage is the fast switch time between frequencies. The challenge is determining how register and memory state are managed among two separate pipelines.

- *Custom-fit processor with variable-depth pipeline*. The system has only a single voltage level, tailored to the speculative frequency (i.e., a low voltage). Therefore, we cannot rely on increasing the voltage to support a higher frequency. Instead, the number of pipeline stages can be doubled. Additional pipeline latches are placed between existing pipeline latches. The extra pipeline latches are normally transparent but can be activated when the processor switches to the recovery frequency. There are two advantages. First, using only a single voltage level makes it easier to verify the design. Second, frequency can be switched a lot faster than voltage, using high-performance phase-locked loops. The challenge is getting good performance out of the deep pipeline mode. The main concern is dependent instructions. For example, intermediate 16-bit results will need to be bypassed among dependent add instructions for performance to scale well when the pipeline depth is doubled. Even more challenging is devising a general strategy for bypassing intermediate results for all instruction types that can. (It is certainly an intriguing research topic and we are actively pursuing it.)

1.3 Related work

There has been much research in the area of dynamic voltage/frequency scaling to minimize power consumption in general-purpose computers [e.g., 5,6,15,18,20]. The general theme is to predict future processor utilization and adjust frequency to reduce power while maintaining performance.

Likewise, a large body of work exists for scheduling real-time tasks on variable frequency/voltage processors to minimize power consumption [e.g., 7,8,11,12]. As pointed out by Mossé et. al. [17], most techniques are based on worst-case estimates of task execution times and work within those constraints (although some techniques exploit variations in task execution times [e.g., 19]).

The closest related work we are aware of is that by Mossé, Aydin, Childers, and Melhem [17]. Like this paper, their work directly addresses the inefficiency of designing real-time systems based solely on *worst-case execution time* (WCET). A task is divided into sub-tasks, which provide periodic power management points. As sub-tasks complete, frequency/voltage are adjusted for remaining sub-tasks based on how much time has *actually* elapsed up to this point. The key is using the *current time* as a basis for frequency selection, which is a summary of actual behavior rather than worst-case behavior.

The main difference is our method, like traditional WCET-based design and unlike the method of Mossé et. al., is not dynamic. The novel contribution is augmenting static analysis with simulation. That is, we use a static design approach based on both the worst-case scenario (for the recovery frequency) and the *simulated-worst-case* scenario (for the speculative frequency). We do not monitor the current time to dynamically re-compute the frequency (it is only monitored to detect mispredictions). Instead, we propose a static approximation of actual elapsed execution time — simulated-worst-case execution time (this is the first summation term in EQ 3, described in Section 2.1).

A dynamic approach is certainly more flexible and can precisely track small changes in the frequency demands of an application. Our approach targets the "low-hanging fruit" — the large gap between guaranteed-correct worst-case bounds and worst-case behavior that occurs in practice. The difference in philosophies is illustrated in Figure 1. We view clock frequency as a *redundant performance precaution* and draw an analogy between transient hardware faults and transient performance faults.

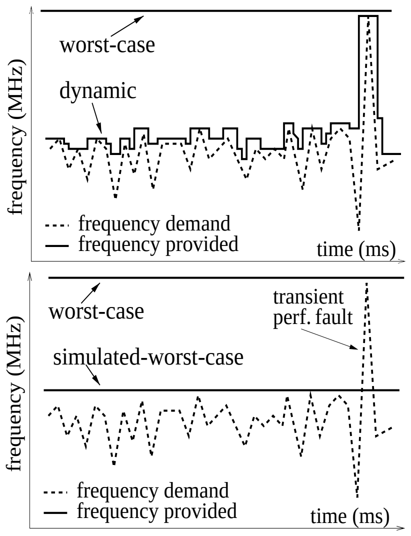


FIGURE 1. Precise tracking (top) vs. our approach (bottom).

There are other differences with prior work as well.

- By targeting the design for the simulated-worst-case, instead of precisely tracking frequency demands, there is less reliance on very fast frequency/voltage switching support. As described in Section 2.1, the frequency is switched at most twice because only a single misprediction is allowed in a task. Furthermore, our experiments include overhead for frequency switching, and the conclusion is that the perceived deadline is shortened by the amount of overhead.

- Run-time overhead is further reduced by not dynamically re-computing frequencies as the task progresses. Code snippets inserted at the end of sub-tasks simply check for mispredictions.
- The equations and methods for statically deriving the speculative and recovery frequencies are based on discrete frequencies, unlike prior work that uses continuous speed settings.
- Our methods do not assume performance scales linearly with frequency. Memory latency is a classic example of an irreducible component of execution time. Our experiments illustrate the importance of modeling non-linear behavior. For example, the result that shorter deadlines result in larger frequency reductions is attributed in part to the irreducible cache miss component.

2. Real-time system design using variable frequency

The proposed method requires static (compile-time or programming-time) and dynamic (run-time) support. The speculative and recovery frequencies are derived via static analysis and simulation. For static analysis, this paper contributes simple intuitive equations that build upon whatever traditional worst-case real-time program analysis is already available (which ranges from naive conservative estimation, to programmer-involved estimation, to intelligent automated estimation). Run-time support consists of a hardware cycle counter and short code snippets inserted at the beginning and end of each sub-task to check for transient performance faults.

2.1 Statically deriving frequencies

A real-time task is initiated by an interrupt, a real-time scheduler that manages a task queue, or a number of other methods. In any case, once initiated, it must complete before a prescribed deadline, as shown in Figure 2.

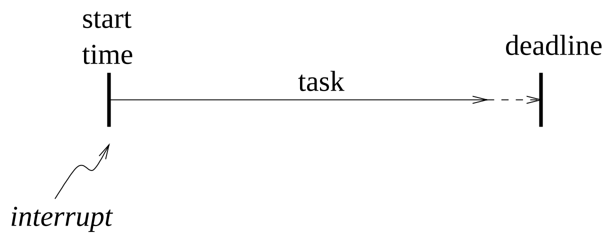


FIGURE 2. Timeline of a task.

In traditional real-time system design, the *worst-case execution time* (WCET) of the task is statically estimated, either manually by the programmer or automatically using a WCET estimation phase in the compiler [9]. The inputs to WCET analysis are the microarchitecture specification (pipeline details, cache and predictor parameters, etc.) and the real-time program. WCET analysis produces an upper bound on the number of cycles required by the task. Correct analysis never underestimates WCET (otherwise the deadline could be missed), and good analysis also minimizes overestimation of WCET. Once WCET is known, a lower bound on the frequency of the processor can be derived (frequency, along with other design considerations, affects the choice of embedded microprocessor used in the system). The amount of over-design implicit in the frequency depends on how much WCET is overestimated.

To enable the new method for hedging microarchitecture performance, the task is partitioned into multiple smaller sub-tasks, as shown in Figure 3. The number and nature of the sub-tasks is arbitrary and entirely up to the designer. For example, the sub-tasks may be different instances of the same region of code, or different regions. Or, the real-time application may already define multiple tasks that run in a predictable sequence and, instead of having individual deadlines, are subject to an overall deadline in combination. In this case, the already-defined group of asymmetric tasks serve as a convenient starting point for sub-task selection.

Before proceeding with the analysis, notation is defined below.

- *T*: Execution time in seconds, *not* cycles. Using cycles is confusing because frequency may affect the number of cycles. Most notably, main memory access time in nanoseconds is usually fixed, and the number of cycles to access main memory increases as frequency increases.
- *s*: The number of sub-tasks.
- *i*: Denotes sub-task *i*. (Likewise, *j* and *k* denote sub-tasks *j* and *k*, respectively.)
- *WC*, *SWC*, *AC*: These denote different scenarios. *WC* stands for worst-case scenario determined by WCET analysis. *SWC* stands for simulated-worst-case scenario observed in practice. *AC* stands for actual-case scenario, i.e., this is what actually happens at run-time for a particular instance of the sub-task. For example, program analysis may show that *WC* is: load #1 always misses and load #2 always misses. Simulations for a set of trials may show that *SWC* is: load #1 always hit and load #2 can miss. A particular dynamic instance of the sub-task may reveal *AC* is: load #1 hit and load #2

hit (*WC* is pessimistic for both loads, *SWC* is pessimistic for load #2).

- f_{wc} : This is the minimum clock frequency needed to meet the deadline, as determined by conventional worst-case analysis (*WC*). I.e., f_{wc} corresponds to conventional over-design.
- f_{spec} : This is the speculative frequency, less than f_{wc}. Sub-tasks are expected to meet their checkpoints when run at f_{spec}, but are not guaranteed to.
- f_{rec} : This is the recovery frequency that guarantees remaining sub-tasks complete before the final deadline, in spite of the fact that the current sub-task missed its checkpoint.
- $T_{i,WC,f}$: Execution time of sub-task *i* under worst-case conditions (see *WC* above) and with the processor running at frequency *f*.
- $T_{i,SWC,f}$: Execution time of sub-task *i* under simulated-worst-case conditions (see *SWC* above) and with the processor running at frequency *f*.
- $T_{i,AC,f}$: Execution time of sub-task *i* under actual conditions (see *AC* above) and with the processor running at frequency *f*.

Note that all three derived frequencies — f_{wc} , f_{spec} , and f_{rec} — are global parameters, that is, they are the same for all sub-tasks.

Inputs to the analysis are (1) the task deadline, (2) sub-tasks, (3) a microarchitecture description, (4) frequencies supported by the microprocessor, and (5) measured worst-case execution times for all sub-tasks and all frequencies (provided by simulation). In practice, a separate microarchitecture description for each frequency is required because main memory latency, in cycles, depends on frequency. Possibly other aspects of the pipeline depend on frequency, too.

The compiler, using state-of-the-art worst-case analysis to minimize pessimism, computes $T_{i,WC,f}$ for all sub-tasks *i* and supported frequencies *f*. It uses items (1)-(4), above, to perform the analysis. The fifth item (5), above, directly provides $T_{i,SWC,f}$ for all sub-tasks *i* and supported frequencies *f*.

The first expression (EQ 1) computes the over-design frequency, f_{wc}. The new speculative technique does not require f_{wc}, however, it provides a basis for comparison. EQ 1 satisfies the overall real-time constraint of the system: the sum of the execution times of all sub-tasks under worst-case conditions must meet the deadline.

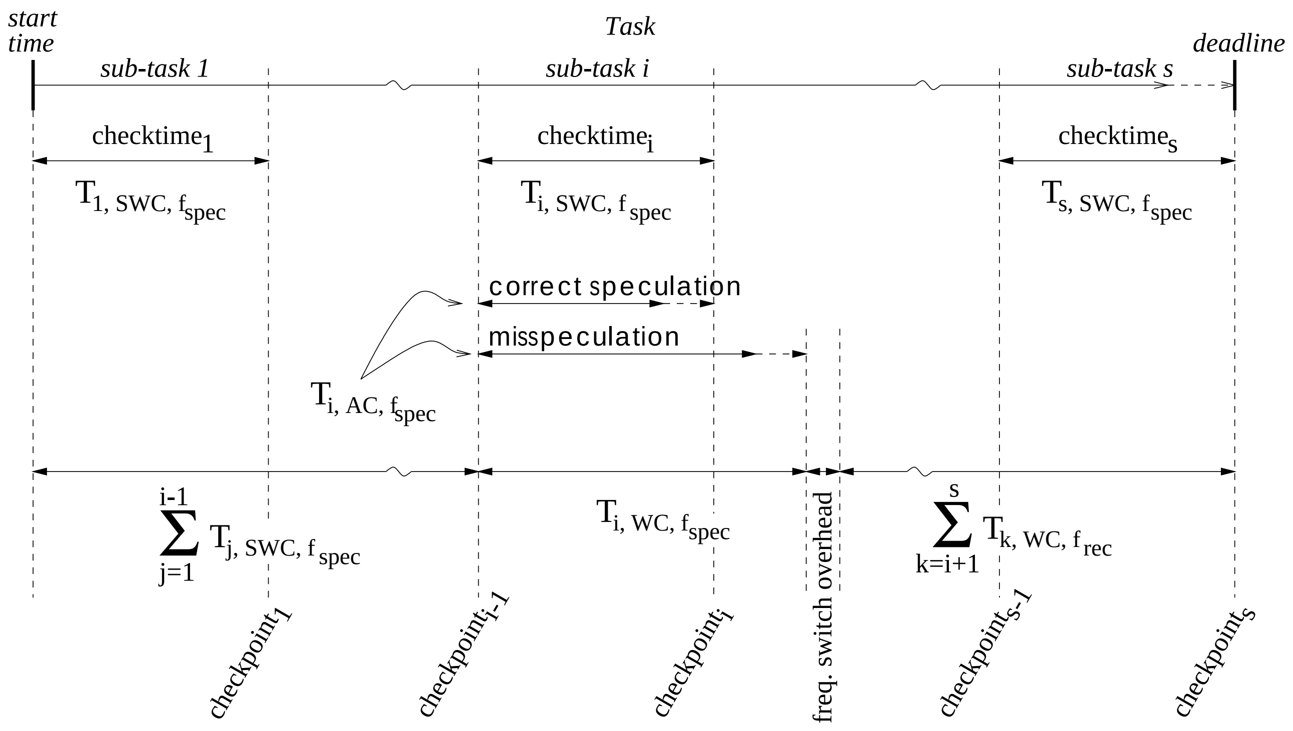


FIGURE 3. Timing of sub-tasks.

$$\sum_{i=1}^{s} T_{i,WC,f_{wc}} \leq deadline \qquad \text{(EQ 1)}$$

To solve for f_{wc}, $T_{i,WC,f}$ for all sub-tasks are substituted into EQ 1, starting with the lowest frequency and increasing frequency until the inequality is satisfied. The minimum frequency that satisfies the above expression gives us f_{wc}.

In Figure 3, checkpoint$_i$ is the expected end time of sub-task$_i$ and the expected start time of sub-task$_{i+1}$. All checkpoints are relative to the origin of the overall task. For analysis, it is convenient to define the period of time between adjacent checkpoints, called a *checktime*. *Checktime$_i$* is the time between checkpoint$_{i-1}$ and checkpoint$_i$, as shown in Figure 3.

The second expression below (EQ 2) simply sets the checktime of a sub-task equal to its simulated-worst-case execution time at the speculative frequency f_{spec}. This means the sub-task is *predicted* to not overrun its checkpoint if the microprocessor uses frequency f_{spec} , and the basis for this prediction is simulation.

$$checktime_i = T_{i,SWC,f_{spec}} \qquad \text{(EQ 2)}$$

Of course, the simulated-worst-case (*SWC*) is not provably the true worst-case, and it is possible for the sub-task's actual execution time at the speculative frequency to exceed its checktime (if the *AC* scenario lies somewhere between the *SWC* and *WC* scenarios).

The actual execution time of speculative sub-task *i* is $T_{i,AC,f_{spec}}$, which is unknown until run-time, and is either less than *checktime$_i$* for correct speculation or greater than *checktime$_i$* for misspeculation. This is shown in Figure 3 for sub-task$_i$.

In the worst case, misspeculation results in an execution time of $T_{i,WC,f_{spec}}$, which is always greater than $checktime_i = T_{i,SWC,f_{spec}}$ because *WC* is worse than *SWC*. (Refer again to Figure 3, sub-task$_i$.)

The simplest approach to guarantee that this timing error does not propagate all the way to the end of the task is to not speculate any remaining sub-tasks. Remaining sub-tasks are clocked at the higher recovery frequency f_{rec}. We assume there is a fixed overhead to switch clock frequencies, as shown in Figure 3. An interesting aspect of our approach is that frequency changes at most two times for a task (from f_{spec} to f_{rec} when there is a misprediction, and then back to f_{spec} at the end of the task to prepare for the next task), which minimizes overhead. The implication is the microprocessor does not necessarily need to provide very fast frequency switching.

To ensure correct recovery, we have to assume that (1) speculative sub-task$_i$ started no earlier than checkpoint$_{i-1}$, (an interesting corollary is that we know sub-task$_i$ started *no later* than checkpoint$_{i-1}$, because no prior sub-task misspeculated), (2) speculative sub-task$_i$ misses its checkpoint by the largest margin possible (execution time is $T_{i,WC,f_{spec}}$), and (3) the worst-case scenario (*WC*) occurs for all remaining sub-tasks. The following expression (EQ 3) ensures that the deadline is met in spite of a single interim microarchitecture failure (there can be at most only one failure in the task), and is also depicted at the bottom of Figure 3.

(EQ 3)

$$\sum_{j=1}^{i-1}(T_{j,SWC,f_{spec}}) + T_{i,WC,f_{spec}} + overhead + \sum_{k=i+1}^{s}(T_{k,WC,f_{rec}}) \leq deadline$$

The first expression in the lefthand side of EQ 3 accounts for the maximum possible time consumed by prior, correctly speculated sub-tasks (it is the sum of prior checktimes). The second expression in the lefthand side of EQ 3 accounts for the maximum possible time consumed by the misspeculated sub-task *i*. The third expression in the lefthand side of EQ 3 accounts for the frequency switching overhead (two switches, as described earlier). Finally, the fourth expression in the lefthand side of EQ 3 accounts for the execution time of remaining sub-tasks at the recovery frequency, and assuming the worst-case scenario for each sub-task. The sum of all these expressions must be less than the deadline.

EQ 3 is solved as follows. We choose a value for f_{spec}, starting at the lowest possible frequency and working upward until a solution is found. For a given f_{spec} attempt, we try to find the minimum f_{rec} that simultaneously satisfies *s* inequalities — there is actually a distinct EQ 3 for each sub-task *i*. If we reach a sub-task *i* for which no f_{rec} satisfies its inequality, then we try the next higher f_{spec} and begin again. Ultimately, the procedure produces a single $\{f_{spec}, f_{rec}\}$ pair, and both frequencies are minimized as much as possible.

2.2 Run-time hardware and software support for detecting and recovering from mispredictions

Hardware provides a cycle counter that can be reset to zero and read by software. The counter is automatically incremented by the microprocessor every clock tick. Also, we assume a control register for switching the clock frequency and querying the current frequency setting.

A code snippet is inserted at the beginning and end of each sub-task, called the prologue and epilogue, respectively. The prologue initializes the cycle counter to 0, in order to measure the number of cycles consumed by the sub-task. The prologue of only the first sub-task initializes to 0 a global variable containing accumulated time in seconds, which is read and updated by each epilogue as described below. Also, the prologue of only the first sub-task sets the processor frequency to f_{spec}, which it usually is anyway (unless the previous task had to recover).

Embedded in each sub-task is its checkpoint time relative to the start time of the task (which was derived by static analysis in Section 2.1). The epilogue checks whether the checkpoint time was met or exceeded, and either does nothing or initiates recovery by switching the processor frequency to f_{rec}.

The execution time of the sub-task is computed by reading the cycle counter and the frequency control register, and dividing the cycle count by the frequency. The result is added to the global variable containing the accumulated time of the task, producing the current time in seconds relative to the start time of the task.

The current time is compared to the sub-task's checkpoint time. If the current time is less, then there is no timing error and the next sub-task may be speculated — the frequency remains f_{spec}. If the current time is greater than the checkpoint time, then this sub-task misspeculated and recovery is initiated — the frequency is set to f_{rec}. Once recovery is initiated, remaining epilogue checks are circumvented.

3. Methodology

3.1 Benchmark (real-time task and sub-tasks)

Standard embedded real-time system benchmarks are not as readily available as SPEC (PCs, workstations, and servers) and TPC (database servers) workloads. However, several universities with research programs in compiler-based WCET estimation have an on-going, organized effort to collect embedded real-time benchmarks [22]. The benchmarks are simple, e.g., sorting, matrix multiplication, fast-fourier transform (FFT), cyclic redundancy check (CRC), etc. Though it is difficult to find standard suites, this variety of benchmark is found in practically all of the WCET papers in the last several years from the Real-Time Systems Symposium. Furthermore, similar benchmark descriptions can be found among the Embedded Microprocessor Benchmark Consortium (EEMBC) testcases [21], although the source code is unfortunately not publicly available.

We use the FFT benchmark downloaded from the C-Lab web site [22] (the FFT version contributed by the Real-Time Research Group at Seoul National University). The FFT benchmark, modified to operate on 1,024 elements, is used as a single sub-task. The real-time task is composed of 16 FFT sub-tasks. The input data for each FFT sub-task differs and is randomly generated, although input data does not significantly impact sub-task execution time because there is no data-dependent control flow.

3.2 Microarchitecture description

The processor has a 7-stage pipeline for ALU and branch instructions — fetch, dispatch (decode and rename), issue, register read, execute, writeback, retire. Instruction execution latencies are similar to those of the MIPS R10K processor. For load and store instructions, the execute stage is expanded into an address generation stage and two stages to disambiguate addresses and access the data cache (therefore, after the address is computed, a data cache hit is two cycles). Instruction issue is out-of-order with a 16-entry reorder buffer. Only 1 instruction is issued per cycle (i.e., not superscalar). The level-1 instruction and data caches, both 8KB direct mapped with 16B lines, are backed directly by main memory. Branch prediction is performed by a 2K-entry bi-modal branch predictor and a 2K-entry branch target buffer.

Main memory latency is always 50 ns, independent of the core's frequency. Supported frequencies and the corresponding memory latency in clock cycles is shown in Table 1 (memory latency in cycles is the ceiling of 50 ns times frequency). There are 11 supported frequencies, from 50MHz to 300MHz, in 25MHz increments.

TABLE 1. Supported frequencies.

frequency (MHz)	main memory latency (cycles)
50	3
75	4
100	5
125	7
150	8
175	9
200	10
225	12
250	13
275	14
300	15

3.3 Generating worst-case execution times

Inputs to the experiments are a deadline, frequencies, simulated-worst-case execution times $T_{i,SWC,f}$ for all sub-tasks and frequencies, and worst-case execution times $T_{i,WC,f}$ for all sub-tasks and frequencies.

Simulated-worst-case execution times are measured by running the FFT sub-task on a detailed microarchitecture simulator. Twenty trials of the sub-task were run for each frequency (with the caches warmed). The longest execution time among the trials of a given frequency provides $T_{i,SWC,f}$.

Worst-case execution times $T_{i,WC,f}$ are normally generated manually or by a phase of the compiler. Manual analysis is time-consuming for complex pipelines. And, unfortunately, we are not aware of any released WCET estimation tools, and creating one is beyond the scope of this paper (we leave this for future work).

To expedite experiments, a pragmatic simulation-based approach is used to generate $T_{i,WC,f}$. To avoid any misinterpretation, carefully note that the method is artificial and does not generate a guaranteed-correct bound for WCET, because it is based on a finite number of simulation trials. The method is only devised to work around not having a compiler with WCET capability.

To mimic uncertainty in the compiler, execution time is over-estimated by randomly injecting a controlled number of additional data cache misses during the simulation trials. Data cache misses are used only as a typical source of uncertainty (other sources include data-dependent control flow, branch prediction, etc.). A miss is injected by converting a cache hit to a cache miss, with some probability. Three probabilities are experimented with — 10%, 30%, and 50% — resulting in over-estimated worst-case execution times called $T_{i,WC10,f}$, $T_{i,WC30,f}$, and $T_{i,WC50,f}$, respectively. If no artificial cache misses are injected (i.e., 0% probability), we simply get the simulated-worst-case execution time $T_{i,SWC,f}$ as before.

We feel injecting cache misses more accurately represents how estimation tools inflate execution time than simply multiplying execution time by an inflation factor. First, it is the *scenario* that estimation tools inflate (e.g., static load #1 misses all the time). Second, memory latency (in cycles) varies with frequency. We have found that injecting additional cache misses has a different impact at 50MHz than at 300MHz. The only way to model such non-linear effects is by simulating the cache misses, not multiplying execution time by a constant factor for all frequencies.

The microarchitecture simulator and benchmark binary are based on the Simplescalar toolset [2]. The Simplescalar compiler is gcc-based and the ISA is MIPS-like. The FFT sub-task is embedded in a benchmark wrapper, which measures the number of cycles consumed by each sub-task trial. This is done via new system calls for resetting and querying the simulator cycle counter. A separate simulation is performed for each frequency (because number of cycles to access main memory changes with frequency, as shown in Table 1), and for each degree of over-estimation (0%, 10%, 30%, and 50%). The resulting execution times $T_{i,SWC,f}$ (0%), $T_{i,WC10,f}$ (10%), $T_{i,WC30,f}$ (30%), and $T_{i,WC50,f}$ (50%) are shown in Figure 4, in units of milliseconds.

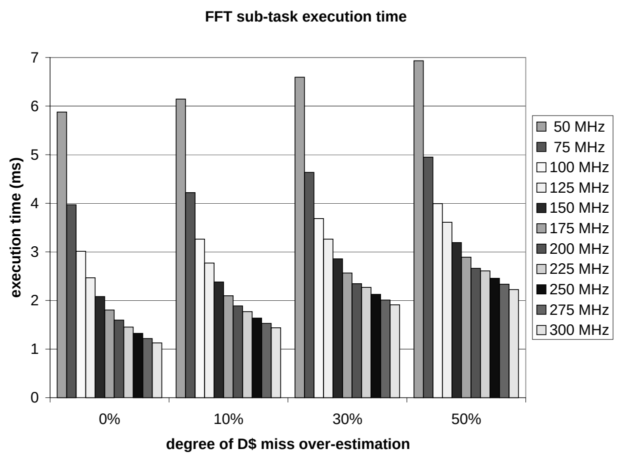


FIGURE 4. FFT sub-task execution time vs. frequency, for various levels of over-estimation

3.4 Automated solver

A tool was developed that solves EQ 1 for f_{wc} and EQ 3 for $\{f_{spec}, f_{rec}\}$, given (1) a deadline, (2) the number of sub-tasks, (3) the number of frequency levels, (4) $T_{i,WC,f}$ for all sub-tasks and frequencies, and (5) $T_{i,SWC,f}$ for all sub-tasks and frequencies. The latter values ($T_{i,WC,f}$, $T_{i,SWC,f}$) were generated in Section 3.3 and are shown in Figure 4. In our benchmark, all sub-tasks i are identical and have the same $T_{i,WC,f}$ and $T_{i,SWC,f}$, but the tool supports sub-tasks that are not identical. The solvers for EQ 1 and EQ 3 are 16 and 43 lines of code, respectively (includes comments).

4. Experiments

The graphs in Figure 5 plot four different frequencies as a function of task deadline, for each of the three worst-case estimation models (WC10, WC30, and WC50). The over-designed frequency, f_{wc} , was derived by solving EQ 1. The speculative and recovery frequencies, f_{spec} and f_{rec} , respectively, were derived by solving EQ 3.

A fourth frequency, *opt* (short for *optimum*), is an ideal lower bound for f_{spec}. We know that none of the *simulated* sub-tasks miss their checkpoints because checkpoints are based on the simulation trials. Knowing this ahead of time, the misspeculation and recovery terms in EQ 3 can be removed to produce a better f_{spec} (EQ 3 is left with only the first term, in which every sub-task meets its checkpoint). Of course, *opt* is based on information that is not known ahead of time in a real system; it is only valid as a measuring stick for f_{spec}.

The first observation is that there is significant speculation opportunity for all of the estimation models. That is, there is a large reduction in frequency between the over-designed frequency f_{wc} and the speculative frequency f_{spec}. For example, for a deadline of 40 ms, there is a frequency reduction of 25 MHz (150 MHz down to 125 MHz) for WC10, 50 MHz (200 MHz down to 150 MHz) for WC30, and 100 MHz (250 MHz down to 150 MHz) for WC50.

The second observation is that the frequency reduction is larger for worse estimation models. Notice the gap between the f_{spec}/*opt* curves and the f_{rec}/f_{wc} curves grows progressively from WC10 to WC30 to WC50. At 40 ms, the frequency reduction of WC50 is two times that of WC30 and four times that of WC10. This makes sense — the disparity between actual execution time (*SWC*) and worst-case execution time (*WC*) increases with poorer estimation models, and there is more opportunity for speculation.

Another trend is that f_{spec} tracks *opt* very closely (but it is never below *opt*, as expected). So, the method for guaranteeing the speculative frequency is quite effective.

Likewise, f_{rec} tracks f_{wc} very closely (but it is never below f_{wc} , as expected). This is also an important result, because it implies that *guaranteeing correct recovery from mispredictions is not much worse than conventional over-design*. The system degrades gracefully to a conventional over-designed system.

Possibly the reason for graceful degradation is that only a single failure is allowed within a task. This can be explained by comparing EQ 1 and EQ 3, which are actually quite similar. The first *i-1* sub-tasks in EQ 3 consume about the same amount of time as the first *i-1* sub-tasks in EQ 1. The speculative EQ 3 sub-tasks run at a lower frequency (f_{spec} vs. f_{wc}) but under more optimistic conditions (*SWC* vs. *WC*), ultimately consuming about the same amount of time as the non-speculative EQ 1 sub-tasks. The single misspeculated sub-task *i* is the only problem, but it is only one exception among many sub-tasks. It is not surprising, then, that the recovery frequency for remaining sub-tasks is close to the over-designed frequency. The last *s-i* sub-tasks in both EQ 1 and EQ 3 run under pessimistic conditions (*WC*) and have about the same amount of time to execute before the deadline (the recovery sub-tasks in EQ 3 have a little less time, because of misspeculated sub-task *i*, but not too much less).

Notice that the speculative frequency is about the same for all of the worst-case estimation models. The top-most graph in Figure 6 shows f_{spec} for WC10, WC30, and WC50. For the most part, the curves overlap. This is an important result because it implies that f_{spec} is relatively insensitive to how much the "compiler" exaggerates worst-case execution time. That is, the degree of pessimism does not limit our ability to guarantee a low speculative frequency. It is actual execution time and not worst-case execution time that dictates f_{spec}.

The exact opposite trend is observed for recovery frequency, shown in the bottom-most graph in Figure 6. Worse estimation models require a higher recovery frequency. For example, for a 40 ms deadline, f_{rec} for WC50 (275 MHz) is nearly twice that of WC10 (150 MHz). Recovery is the insurance policy that covers speculation, and guarantees must be based on provably correct worst-case bounds. So, f_{rec} is naturally sensitive to worst-case execution time.

Finally, referring back to Figure 5, speculation opportunity tends to increase with tighter deadlines. That is, frequency reduction increases with tighter deadlines. For example, the difference between f_{spec} and f_{wc} for the best estimation model, WC10, is constant at about 25 MHz for most of the graph, but a change occurs at 28 ms. The difference between f_{spec} and f_{wc} increases to 50 MHz at 28 ms and 75 MHz at 26 ms.

The same trend is observed for the other models. For WC50, the difference between f_{spec} and f_{wc} is 75 MHz at 50 ms and reaches as high as 150 MHz at 37 ms.

Frequency delivers diminishing performance returns because one component of execution time, memory latency, is not reduced with higher frequency. Tighter deadlines require more performance. Meanwhile, frequency is progressively less effective at generating performance. So, f_{wc} increases non-linearly to compensate for diminishing performance returns, widening the gap between f_{wc} and f_{spec} (f_{spec} increases slower because it is based on SWC, which has a smaller memory component).

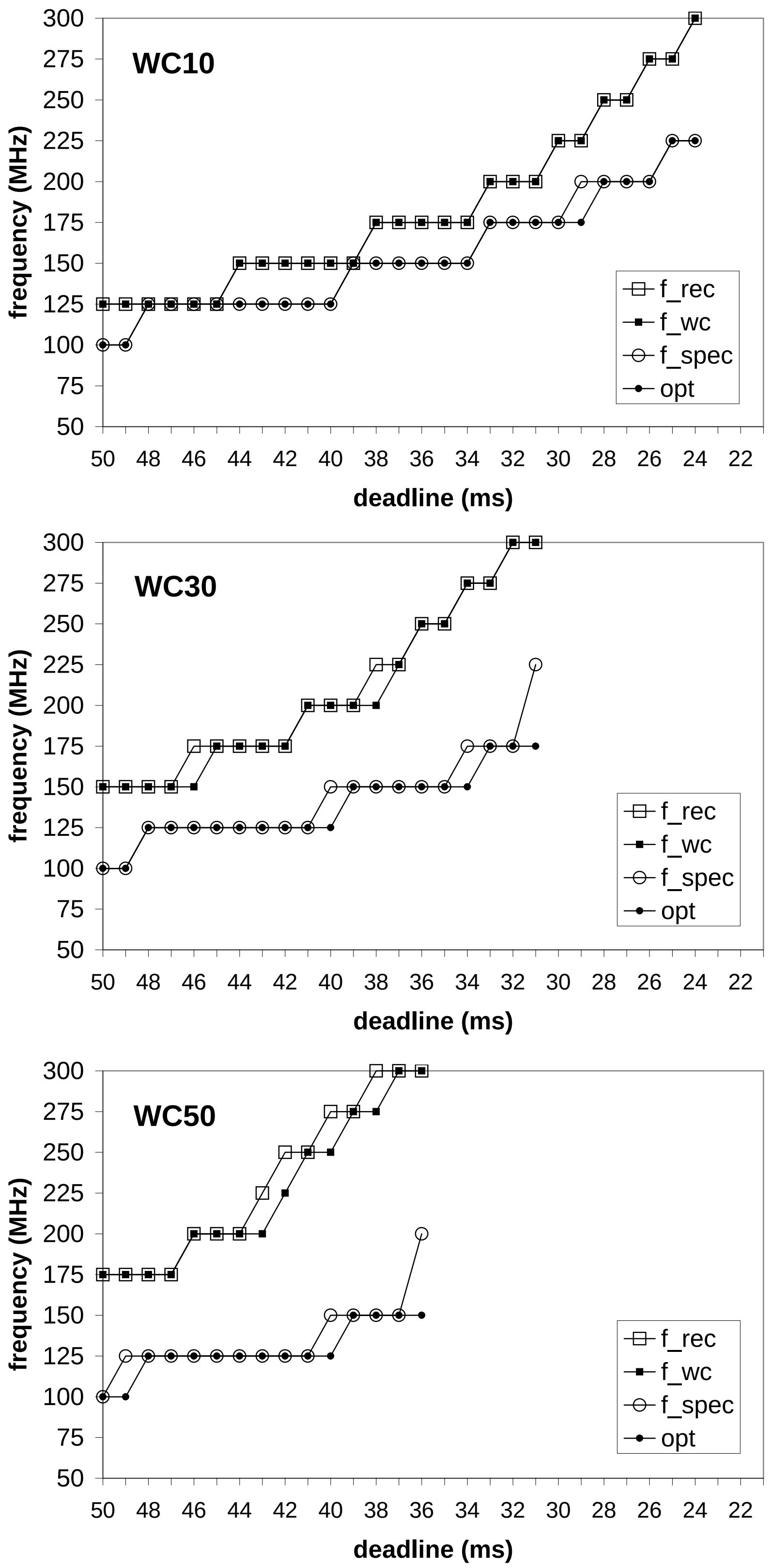
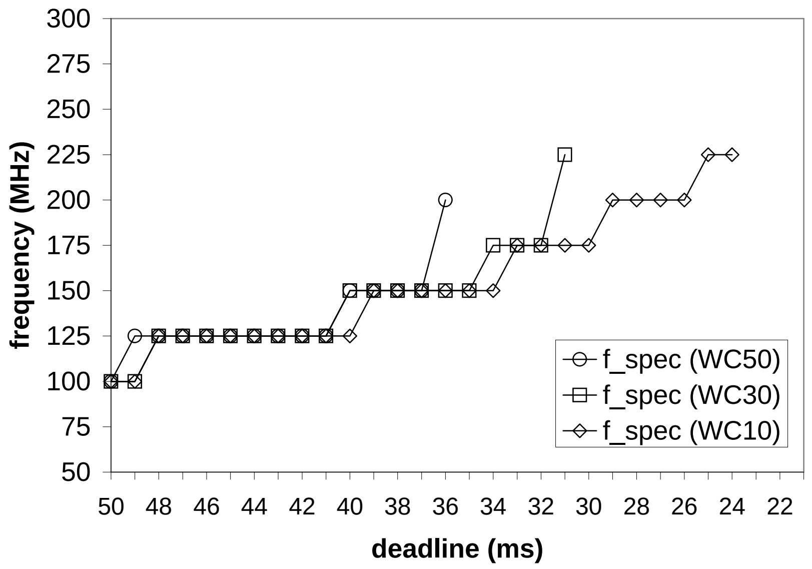


FIGURE 5. Frequencies for each of the worst-case estimation models.



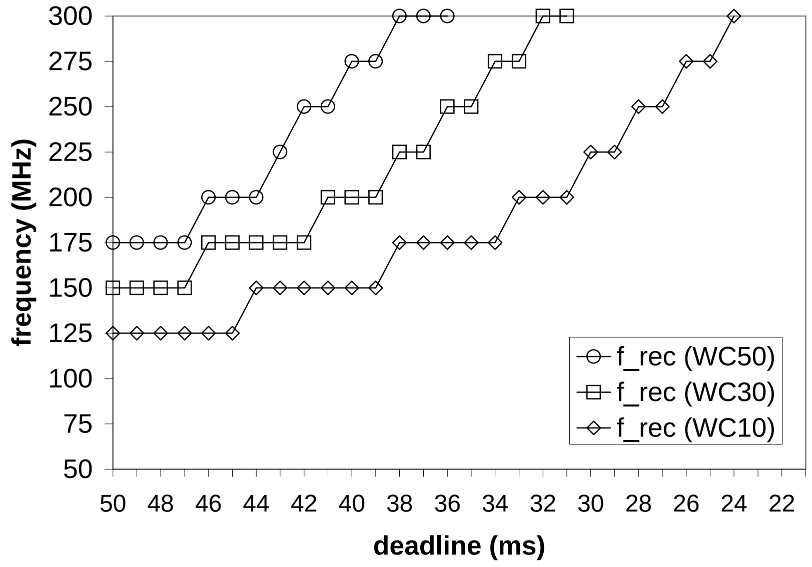


FIGURE 6. Comparing the speculative frequencies (top-most graph) and recovery frequencies (bottom-most graph) of different worst-case estimation models.

Frequency switching overhead was not accounted for in the previous experiments. The graph in Figure 7 shows the difference in f_{spec} with and without a 1 ms switching overhead, for the WC10 model. The curve with overhead is identical to the one without, except it is shifted to the left by 1 ms. Essentially, a 1 ms overhead reduces the perceived deadline by that amount (e.g., a 40 ms deadline looks like a 39 ms deadline), which is consistent with EQ 3. The same result was observed for WC30 and WC50.

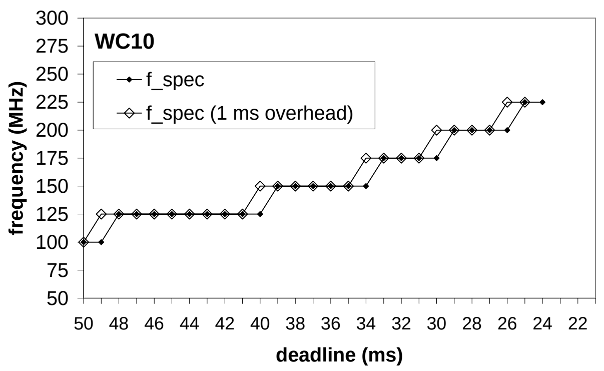


FIGURE 7. Impact of switching overhead.

5. Summary and future work

High-performance microarchitecture techniques such as pipelining, caching, and branch prediction are making their way into embedded processors. Unfortunately, worst-case analysis for real-time systems underestimates microarchitecture contributions because it is difficult to guarantee performance of complex pipelines. Ultimately, the designer must turn to clock frequency as a redundant, reliable source of performance. Over-designing clock frequency apparently defeats the purpose of also adding hardware enhancements.

We propose that simulation coupled with traditional WCET analysis can resolve this paradox. Simulated-worst-case bounds determine a speculative frequency and guaranteed-correct worst-case bounds determine a recovery frequency. A sub-task is expected to meet its checkpoint at the speculative frequency. If it does not, a transient performance fault is detected and the processor recovers by running remaining sub-tasks at the recovery frequency.

Experiments demonstrate frequency reduction of up to 100 MHz for a peak 300 MHz processor. Other key results: benefits increase with more conservative WCET analysis; benefits increase with tighter deadlines; and the recovery frequency is close to the frequency produced by conventional worst-case analysis, indicating graceful degradation in the presence of faults.

Future work includes evaluating the technique for more complex tasks, studying sub-task selection, integrating/developing WCET compilers to generate worst-case execution times, and exploring custom-fit processors as a platform.

Acknowledgments

The author thanks Alex Dean, Zach Purser, and Anu Vaidyanathan for helpful discussions. This research was supported by generous funding and equipment donations from Intel, Ericsson, and by NSF CAREER grant No. CCR-0092832.

References

[1] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. *15th Real-Time Systems Symposium*, 1994.

[2] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The Simplescalar Toolset. Technical Report CS-TR-96-1308, Computer Sciences Department, University of Wisconsin - Madison, July 1996.

[3] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. *21st Real-Time Systems Symposium*, Nov. 2000.

[4] J. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: Letting applications define architectures. Technical Report HPL-96-144, HP Labs, Oct. 1996.

[5] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. *1st Int’l Conference on Mobile Computing and Networking*, Nov. 1995.

[6] D. Grunwald, P. Levis, C. Morrey III, M. Neufeld, and K. Farkas. Policies for dynamic clock scheduling. *Symp. on Operating Systems Design and Implementation*, Oct. 2000.

[7] I. Hong, M. Potkonjak, M. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. *Int’l Conference on Computer-Aided Design*, Nov. 1998.

[8] I. Hong, G. Qu, M. Potkonjak, and M. Srivastava. Synthesis techniques for low-power hard real-time systems on variable voltage processors. *19th Real-Time Systems Symposium*, Dec. 1998.

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

[10] S.-K. Kim, R. Ha, and S. L. Min. Analysis of the impacts of overestimation sources on the accuracy of worst case timing analysis. *20th Real-Time Systems Symposium*, Dec. 1999.

[11] C. Krishna and Y. Lee. Voltage clock scaling adaptive scheduling techniques for low power in hard real-time systems. *6th Real-Time Technology and Applications Symposium*, May 2000.

[12] Y. Lee and C. Krishna. Voltage clock scaling for low energy consumption in real-time embedded systems. *6th Int’l Conf. on Real-Time Computing Systems and Applications*, Dec. 1999.

[13] Y. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. *16th Real-Time Systems Symposium*, 1995.

[14] Y. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. *17th Real-Time Systems Symposium*, 1996.

[15] J. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. *Proceedings of the ACM SIGMETRICS 2001 Conference*, June 2001.

[16] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. *20th Real-Time Systems Symposium*, Dec. 1999.

[17] D. Mossé, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. *Workshop on Compilers and Operating Systems for Low Power*, Oct. 2000.

[18] T. Pering, T. Burd, and R. Brodersen. The simulation of dynamic voltage scaling algorithms. *Symp. on Low Power Electronics*, 1995.

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

[20] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. *1st Symp. on Operating Systems Design and Implementation*, Nov. 1994.

[21] “EEMBC: Embedded Microprocessor Benchmark Consortium,” http://www.eembc.org.

[22] “C-lab: WCET Benchmarks,” http://www.c-lab.de/home/en/download.html.