

# FAST: Frequency-Aware Static Timing Analysis

Kiran Seth, Aravindh Anantaraman, Frank Mueller and Eric Rotenberg \*

Departments of CS/ECE and Center for Embedded Systems Research  
North Carolina State University, Raleigh, NC 27695-7534

E-mail: mueller@cs.ncsu.edu, phone: +1.919.515.7889

## Abstract

*Power is a valuable resource in embedded systems as the lifetime of many such systems is constrained by their battery capacity. Recent advances in processor design have added support for dynamic frequency/voltage scaling (DVS) for saving power. Recent work on real-time scheduling focuses on saving power in static as well as dynamic scheduling environments by exploiting idle and slack due to early task completion for DVS of subsequent tasks. These scheduling algorithms rely on a priori knowledge of worst-case execution times (WCET) for each task. They assume that DVS has no effect on the worst-case execution cycles (WCEC) of a task and scale the WCET according to the processor frequency. However, for systems with memory hierarchies, the WCEC typically does change under DVS due to frequency modulation. Hence, current assumptions used by DVS schemes result in a highly exaggerated WCET.*

*This paper contributes novel techniques for tight and flexible static timing analysis particularly well-suited for dynamic scheduling schemes. The technical contributions are as follows: (1) We assess the problem of changing execution cycles due to scaling techniques. (2) We propose a parametric approach towards bounding the WCET statically with respect to the frequency. Using a parametric model, we can capture the effect of changes in frequency on the WCEC and, thus, accurately model the WCET over any frequency range. (3) We discuss design and implementation of the frequency-aware static timing analysis (FAST) tool based on our prior experience with static timing analysis. (4) We demonstrate in experiments that our FAST tool provides safe upper bounds on the WCET, which are tight. The FAST tool allows us to capture the WCET of six benchmarks using equations that overestimate the WCET by less than 1%. FAST equations can also be used to improve existing DVS scheduling schemes to ensure that the effect of frequency scaling on WCET is considered and that the WCET used is not exaggerated. (5) We leverage three DVS scheduling schemes by incorporating FAST into them and by showing that the power consumption further decreases. To the best of our knowledge, this study of DVS effects on timing analysis is unprecedented.*

## 1. Introduction

Limitations on the lifetime of embedded devices, particularly battery-powered mobile devices, have resulted in advances in embedded architecture to extend the lifetime of devices. Microprocessor designs support dynamic adjustment of processing speed to prolong battery life. Generally, two techniques are employed in unison. On one side, dynamic frequency scaling allows the speed of instruction execution to change during the operation of a device. On the other side, dynamic voltage scaling modulates the level of the supply voltage upon demand. Generally, both schemes, referred to as DVS in the following, work hand in hand: When the frequency is lowered by a certain degree, the voltage can be also be reduced to a lower level. Furthermore, both scaling techniques impact the power consumption of a device: power scales linearly with the frequency and quadratically with the voltage. Hence, considerable power savings may result in a concerted approach of dynamic frequency and voltage scaling [6].

Real-time systems are particularly well-suited to profit from DVS. Due to periodic task execution, it is generally not feasible to utilize the range of sleeping modes that modern processors offer. Tasks are invoked frequently (on a periodic basis in the order of a few milliseconds). The time to enter a sleep mode (and the later wakeup time) is in the order of tens of milliseconds, which generally matches the order of magnitude of a real-time task's period. Hence, suspension in sleep modes is not a viable option for real-time systems. But real-time systems often have task sets that underutilize the processor. Hence, reducing the frequency of execution while still meeting deadlines through DVS is a viable option resulting in considerable power reduction.

Recently, a number of hard real-time DVS scheduling schemes have been studied, ranging from compiler support [16] over numerous static scheduling approaches [10, 19] to dynamic methods [19, 2, 9]. All of these approaches have their own merits in that they provide a solution suitable to certain systems depending on scheduling methods, utilization bounds of the task sets and architectural properties, such as scaling overhead.

Any DVS scheduling scheme is subject to the same constraints as other hard real-time systems: The worst-case execution time (WCET) of a task has to be known *a priori*, *i.e.*, safe bounds on a task's execution time have to be ob-

---

\* This work was supported in part by NSF grants CCR-0208581, CCR-0310860 and CCR-0312695.

tained. Prior work on static timing analysis provides the means to derive relatively tight WCET bounds for simple embedded architectures, which are provably safe. A number of research groups have addressed various issues in the area of bounding the WCET of a real-time task. Conventional methods for static analysis have been extended from unoptimized programs on simple CISC processors to optimized programs on pipelined RISC processors, and from uncached architectures to instruction and data caches [18, 14, 12, 17, 23, 13]. The challenge of static timing analysis is to provide not only safe but also tight bounds on the WCET in order to impose a high enough processor utilization. These analysis approaches result in tight bounds for deterministic microarchitectures with simple components.

In the context of DVS, static timing analysis is generally assumed to remain valid with frequency scaling. The conjecture is that reducing a processor's frequency still results in the *same* number of cycles of execution for a task. Hence, considering the processor frequency should suffice to derive safe WCET bounds. However, this simplistic view generally *does not hold* for any realistic architectures. Consider the impact of memory references. Any instruction or data reference that is resolved through a main memory access operates at external bus frequency. But bus frequencies generally diverge from internal processor frequencies, and they do *not* scale at the same rate as DVS scaling does. *E.g.*, the first generation Compaq Ipaq has a StrongArm microprocessor (SA-1110) that scales at 8 frequencies but only supports two different external bus frequencies.

In short, when static timing analysis is applied in the context of DVS, tightness and safety assumptions may no longer hold: WCET bounds may either not be tight (considerable overestimation upon fast memory operations for lower processor frequencies) or are no longer safe (underestimation potentially leading to missed deadlines upon a reduced data bus frequency). As a result, the memory latency also has to be adjusted to discrete values according to dynamic settings for execution frequencies and memory latencies. Instead of obtaining one discrete WCET through static timing analysis, different values for each processor frequency / bus frequency pair would have to be obtained. While this may still be a feasible approach for a static schedule and for a small number of such frequency pairs, it becomes infeasible for dynamic scheduling paradigms or a large number of frequency pairs. For certain scheduling approaches that exhibit intra-task DVS, such a static approach becomes impossible if tight bounds for the WCET are to be determined since the point of frequency changes during task execution is typically unknown at static time, *e.g.*, due to dynamic scheduling, preemption and early completion.

The contribution of this paper is to remedy this problem by promoting a new methodology for frequency-aware static timing analysis (FAST). Instead of obtaining a WCET

bound for each frequency pair, FAST takes static timing analysis to a novel level suitable for dynamic scheduling. FAST expresses WCET bounds as a parametric term whose components are frequency-sensitive parameters. On the one side, cycles are interpreted in terms of the processor frequency; on the other hand, memory accesses are expressed in terms of the memory latency overhead due to the external bus speed. This parametric expression of the WCET allows one to determine on-the-fly the WCET for a given frequency pair. This is particularly appealing when scheduling decisions occur dynamically and when the number of frequency pairs becomes large, such as is the case with state-of-the-art processors with fine-grained frequency settings.

In the following, we detail the technical innovations necessitated by DVS to ensure that safe and flexible WCET predictions may be obtained. We provide motivating examples, discuss the design of our FAST analysis tool, and we show the feasibility of our approach in a set of experiments that demonstrate flexibility and competitiveness while still providing tight bounds on the WCET. Related as well as future work and a summary conclude our contributions.

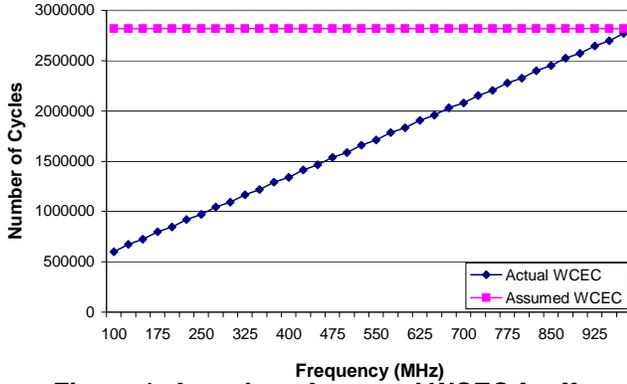
## 2. Effects of frequency scaling on WCET

In this section, we motivate the need for a parametric frequency model and assess the challenges of supporting this novel model in a static timing analysis tool. We also describe the parametric frequency model in detail, and we illustrate the key features in examples.

### 2.1. Motivation

Real-time systems that use DVS-based scheduling scale the WCET assuming that the WCEC remains constant even with a change in the frequency. This assumption holds for systems where the memory latency can scale with processor frequency (systems with on-chip memory) In contrast, for a system where the memory latency does not scale with processor frequency (systems with dynamic memory and memory hierarchies), the WCEC of a task *does not* remain constant when the frequency is scaled since an increase in the frequency typically increases the number of cycles required to access memory. This behavior is caused by a constant access latency for memory references, regardless of changing processor frequencies. By assuming that the WCEC remains constant, one ignores the fact that the WCEC *reduces* with frequency, which results in WCET overestimations.

Figure 1 depicts results for the C-lab real-time benchmark *fft*, where the actual WCEC for a system with a memory hierarchy is compared to a constant WCEC. The WCEC for the benchmark was calculated for a simple in-order pipeline with instruction and data caches. In this example, it is assumed that the memory access latency is constant. Figure 1 illustrates that the number of WCEC increases proportionally with the processor frequency. This results from an increasing number of wait cycles for a constant time



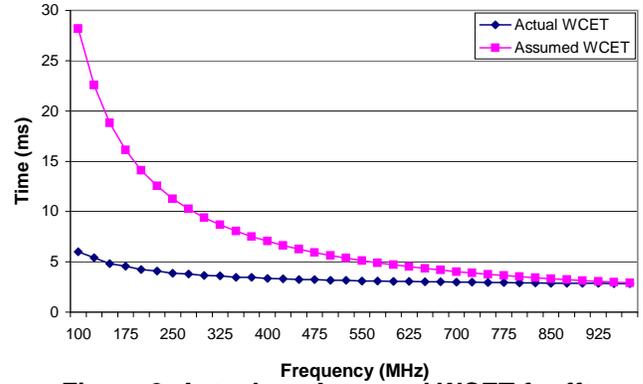
**Figure 1. Actual vs. Assumed WCEC for fft**

memory latency as the frequency increases. The slope of the actual WCEC depends on the number of accesses to main memory (and the latency to frequency ratio). Hence, the slope depends on the number of misses in the instruction and data caches combined. Therefore, the accuracy of paradigms that measure the worst-case behavior of the instruction and data caches not only control the accuracy of the WCEC, but they also affect the accuracy by which the WCEC can be scaled with frequency. Figure 2 depicts the equivalent WCET to the two WCEC curves in Figure 1. The actual WCET depicted indicates the assumption of a constant WCEC independent of frequency modulations result in considerable overestimations of the WCET.

The objective of the work described in this paper is to accurately model the actual WCEC and, thereby, the actual WCET of real-time tasks. We derive a parametric frequency model for this purpose. The model provides WCET bounds that remain tight and accurate throughout any frequency range. The parametric model complements real-time systems employing a DVS-base scheduling scheme, and it is paramount to achieving higher power savings. Ignoring the change in WCEC with frequency results in considerably smaller power savings.

## 2.2. Parametric Frequency Model

Our parametric frequency model can be used for timing analysis with any simple in-order single-issue pipeline. The model is applicable to systems with or without a memory hierarchy. We consider the model in a system with a memory hierarchy in the following, and we contribute solutions to the technical challenges posed. We assume that the system is equipped with an on-chip instruction and data cache and that the main external memory has a constant access latency. To accurately model the WCET in systems with memory hierarchies, we propose a parametric frequency model that captures the effect of frequency scaling accurately by splitting the WCEC of a task into two components. The first component,  $i$ , captures the ideal number of cycles required to execute the task assuming perfect caches. In other words,  $i$  does not scale with frequency. The second component,  $m$ , counts the total number of instruction and data cache misses



**Figure 2. Actual vs. Assumed WCET for fft**

for the task.  $m$  is the part of the WCEC that scales with frequency and depends on the memory access latency. If a system without caches is considered,  $i$  would count the total number of cycles used for non-memory operations while  $m$  would count the total number of memory references. Thus, the WCEC is expressed as follows:

$$WCEC = i + mN \quad (1)$$

where  $N$  is the number of cycles required to access the memory, which depends on the latency of the memory and the frequency of the processor. For a simple pipeline, the WCEC can be easily be converted into the WCET by dividing by the frequency. This frequency model can accurately model the actual WCET because it separates the WCEC into components, one that scales and one that does not scale with processor frequency.

The following examples are presented to show that the parametric model can capture the effects of different sequences of instructions in a task. Only sequences that contain data or instruction cache misses are of concern since they are affected during frequency scaling. A sequence of instructions without any cache misses can be captured exclusively by the  $i$  component and represents a trivial example of our parametric model. For the following examples, let  $N = 10$ , as shown in the figures below. We assume separate instruction and data caches and frequency scaling under our model with an arbitrary simple in-order pipeline.

Consider a sequence of four instructions, as shown in the Figure 3. This instruction sequence is executed in a processor with a simple six-stage in-order pipeline. The pipeline stages are fetch (IF), decode (ID), issue (IS), execute (EX), memory access (MEM) and write-back (WB).

1. In Figure 4, we observe the effects of an instruction cache. Consider instruction B resulting in a miss. While instruction B misses in the instruction cache, all other cache accesses result in hits. Since instructions are stalled till the

- A: add R2, R1, R3
- B: load R4, [M1]
- C: add R2, R1, R4
- D: add R2, R1, R5

**Figure 3. Sample Instruction Sequence**

miss on B is resolved, the number of cycles involved can be separated into two components. With  $i = 9$  and  $m = 1$  in Equation 1, the WCEC is accurately captured by our model as  $WCEC = 9 + 1N$ . Hence, the WCEC is accurately modeled for any value of  $N$  resulting in an accurate WCET regardless of frequencies.

Cycles	1	2	3	4	5	6	-	-	12	13	14	15	16	17	18	19
IF	A	B	B	B	B	B	-	-	B	C	D					
ID		A								B	C	D				
IS			A								B	C	D			
EX				A								B	C	D		
MEM					A								B	C	D	
WB						A								B	C	D

**Figure 4. Ex 1: Instruction cache miss**

2. In Figure 5, we observe the effects of a data cache miss. Instruction B misses in the data cache while all other cache accesses are hits. With  $i = 9$  and  $m = 1$ , the WCEC is again calculated as  $9 + 1N$ . Since the data miss stalls the previous instructions, one can separate the number of cycles required for the memory access. However, had the Instruction C or any other stalled instruction performed any useful work instead of being stalled, a potential for overestimation would occur for the model, *e.g.*, for multi-cycle floating-point operations, branch mispredictions, etc. Any such overestimation results from the overlap of useful cycles with the memory stall. In our model, the  $i$  component counts these useful cycles while the  $m$  component counts data miss. Overlap would not be considered by the model. For example, if instruction C took an extra cycle to execute, the new WCEC would become  $10 + 1N$ . The model does not consider the overlap between the data miss and the extra cycle used by instruction C. A similar problem is also observed in example 1 if the instruction miss overlaps with a high execution latency instruction.

The potential for overestimations implies that the obtained WCET obtained still provides an upper bound on the execution time, albeit not necessarily a tight one. But removing overestimations due to instructions with high execution latencies is non-trivial because instructions may have different execution latencies. Subsequent experiments show that these design choices have a diminishing affect on the tightness of WCET bounds.

3. In Figure 6, we observe the effects of a simultaneous instruction and data cache misses. Instruction B results in a data cache miss while the instruction C results in an in-

Cycles	1	2	3	4	5	6	-	-	16	17	18	19
IF	A	B	C	D								
ID		A	B	C	D							
IS			A	B	C	D	-	-	D			
EX				A	B	C	-	-	C	D		
MEM					A	B	-	-	B	C	D	
WB						A				B	C	D

**Figure 5. Ex 2: Data cache miss**

struction cache miss. All other cache accesses are hits. With  $i = 9$  and  $m = 2$ , the  $WCEC = 9 + 2N$ . The instruction and the data cache misses cannot be serviced together. Hence, instruction B is stalled till instruction C's cache miss is serviced. The model captures all sequences of instructions where a cache miss stalls yet another cache miss. Notice that the two misses in question need not result from consecutive instructions. We observe some overestimation because of overlapping of some work with the miss cycles.

Cycles	1	2	3	4	5	6	-	-	13	14	15	16	-	24	25	26	27
IF	A	B	C	C	C	C	-	-	C	D							
ID		A	B							C	D						
IS			A	B							C	D	-	D			
EX				A	B							C	-	C	D		
MEM					A	B	-	-	B	B	B	B	-	B	C	D	
WB						A									B	C	D

**Figure 6. Ex 3: Instruction + data cache miss**

In the above examples, different combinations of cache misses were considered, which can occur in a simple pipeline. In the presence of these misses, the parametric model accurately captures the worst-case timing behavior for any sequence of instructions. Overestimation is expected when a high execution latency operation overlaps with a miss or when an I-cache miss overlaps with a D-cache miss.

### 3. Timing Analysis

In this section, we describe conventional static timing analysis and briefly contrast the approach to dynamic timing analysis methods. We specify the novel enhancements necessitated by DVS to adapt conventional static timing analysis to a frequency-aware static timing analysis (FAST) tool.

#### 3.1. Static Timing Analysis

Schedulability analysis for hard real-time systems requires that the worst-case execution time (WCET) be safely bounded in order to ensure feasibility of scheduling a task set for a given scheduling policy, such as rate-monotone and earliest-deadline-first scheduling [15]. If the execution time of a task were obtained through dynamic timing analysis based on experimental or trace-driven approaches, these values would not provide a safe bound of the WCET [22]. On the one side, it is difficult to determine the worst-case input set even for moderately complex tasks that would exhibit the WCET, and to perform exhaustive testing over the entire input space is infeasible except for trivial cases. On the other side, even if the worst-case input set was known, the interaction between the software and hardware might cause the task to exhibit its WCET for a different input set. The cause of this behavior is architectural complexity, such as complex pipelines and caching mechanisms.

Static timing analysis is a viable alternative to dynamic timing analysis, and while various static approaches have been studied, we will constrain ourselves to one such toolset without loss of generality [11, 17, 23]. The WCET bounds

obtained by static timing analysis provide a guaranteed upper bound on the computation time of a task. 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. Loop bodies only require a few traversals to determine the worst-case behavior of the entire loop due to an efficient fixed-point approach. As the execution paths are traversed, the behavior of the architectural components along the execution paths is captured. The paths are composed to form loops, functions and ultimately the entire application to calculate both WCEC and WCET.

Figure 7 depicts an overview of the organization of this timing analysis toolset. An optimizing compiler has been modified to produce control flow and branch constraint information as a side effect of the compilation of a source file. The original research compiler VPCC/VPO [3] was replaced by GCC with a Portable Instruction Set Architecture (PISA) backend that interfaces with SimpleScalar. Real-time applications are compiled into assembly code using the GCC PISA-compiler. The control-flow graph and instruction as well as data references are extracted from the assembly code. Upper bounds on the number of iterations performed by loops are provided, a prerequisite for performing static timing analysis. A static instruction 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 of each function. The program’s control-flow graph is then analyzed, and a caching categorization for each instruction and data reference in the program is produced. Separate categorizations are provided for each loop level in which the instructions and data references are contained. The categorizations for instruction references are described in Table 1. Next, the timing analyzer uses the control flow and constraint information, caching categorizations, and machine dependent information (*e.g.*, pipeline characteristics) to calculate bounds on the WCET.

The approach in this paper differs from our prior toolset as follows. Our tool separates static I-cache and D-cache (instruction/data cache) analysis. The D-cache analysis currently lacks sufficiently detailed information about references for the GCC compilation phase, and D-cache analysis does not fully match the SimpleScalar model. The focus of this paper is on enhancing the timing analyzer with respect to the FAST model and PISA instruction set. But since we use our SimpleScalar-based architectural simulation environment [20] to validate our approach, we have to make

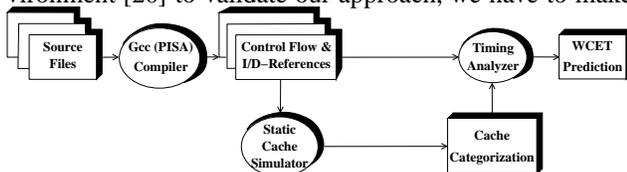


Figure 7. Obtaining Safe WCET Bounds

Cache Category	Definition
always miss	Instruction may not be in cache when referenced.
always hit	Instruction will be in cache when referenced.
first miss	Instruction may not be in cache on 1st reference for each loop execution, but is in cache on subsequent references.
first hit	Instruction is in cache on 1st reference for each loop execution, but may not be in cache on subsequent references.

Table 1. Instruction Categories for WCET

simplifying assumptions about data caches. Specifically, we assume a constant number of data cache accesses to be misses for each application to model compulsory misses. The remaining references are considered to be hits, which models a sufficiently large cache. This simplifying assumption does not affect the design of FAST, *i.e.*, our model supports a more precise static data cache analysis as well.

The timing analyzer uses the control-flow information and loop bounds, caching categorizations, and pipeline description to derive WCET bounds. 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). Path analysis (see below) selects the longest execution path as usual. Once timings for alternate paths in a loop are obtained, a fixed-point algorithm (quickly converging in practice), is employed to safely bound the time of the loop based on the its body’s cycle counts.

The fixed-point approach generally 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 fixed-point, subsequent loop iterations can be safely approximated by this fixed-point timing value. 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 fixed-point approach, the timing analyzer ultimately 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 / caller is not calculated until the times for all of its inner loops / callees 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, eventually reaching the tree’s root (*e.g.*, `main()`). For our purposes, the timing analysis tree provides a convenient method for obtaining WCET for a specific scope, in particular for sub-tasks. From the description in this section, it becomes evident that static timing analysis is non-trivial, even for simple pipelines.

### 3.2. Frequency-Aware Static Timing Analysis

The static timing analysis tool calculates the WCEC for a particular task. However, static timing analysis has to be performed whenever the processor frequency is changed. Re-assessing the WCET bound is paramount to temporal safety since a change in the processor frequency causes a change in the number of cycles required to access the memory since front-side bus frequencies do not scale at all (or at least not at the same rate). Due to the change in memory latency, the WCEC information for different paths changes, which may result in a different worst-case path than before. Our frequency model can be elegantly incorporated into static timing analysis such that it calculates the number of cycles for *each possible worst-case path* in the program. The following technical innovations to the static timing analysis framework support such flexible calculations.

Instead of using the memory access cycles to simulate the sequence of instructions in the pipeline, the ideal number of cycles is calculated assuming all cache accesses to be hits. The instruction and data cache misses are accumulated as a side-effect to compose a first-order polynomial equation describing the WCEC.

Static timing analysis requires different paths through the same node (loop or function) to be compared. The path with the worst WCEC is used as the WCEC for the node. After integrating the frequency model into the framework, one has to compare two equations to determine which one was to result in a larger number of execution cycles. The challenge here is posed by having to consider both equations: One of them has greater WCEC for some range of frequencies while the other has greater WCEC for the rest of the frequency range. Remember that the frequency model is a first-order polynomial. Consider the case where two equations intersect, *i.e.*, both polynomial have a common solution. We propose three approaches to address this problem.

**1.** One can maintain an ordered list of equations and the ranges where subsequent polynomials represent a larger WCEC than previous ones. Since the frequency model is a

first-order polynomial with different slopes, there exists an intersection point constraining the range for each equation.

**2.** Alternatively, a curve-fitting equation could capture the effects of both equations. This obviates the need for maintaining large numbers of equations but increases the complexity of the parametric equation. A higher-order polynomial with strict upper bounds on each base polynomial would provide a relatively close fit. The resulting curve would not be as tight as in case (1) but may suffice if the slopes of the original polynomials do not diverge significantly. This would impose more overhead on dynamic scheduling schemes that have to perform additional arithmetic to evaluate the equation upon any scheduling action.

**3.** Another, easier solution is to declare a valid range of frequencies for the processor. If two equations intersect outside the given range, we simply have to choose the equation that provides the higher WCEC within the valid range. If two equations intersect within this specified range, we use a simple curve-fitting technique through a first-order polynomial that provides a WCEC greater or equal to the values of either of the original equations.

By using one of the above techniques, we ensure that a FAST equation obtained always provides an upper bound on the WCEC of the task, regardless of the chosen frequency. For our FAST framework, we have used the third, the easiest technique to bound FAST equations.

## 4. FAST-DVS Schemes

Most DVS scheduling algorithms use the assumption that the WCEC is constant with frequency when scaling the WCET. By not considering the effect on WCEC during frequency modulation, DVS schemes assume a considerably overestimated WCET. Thus, DVS schemes fail to completely utilize available slack because the scaled WCET is not a tight bound. We have implemented our parametric frequency model as the FAST framework. Parametric equations obtained by FAST can be used in DVS scheduling schemes to ensure that the scaled WCET remains an accurate and tight bound of the execution time for a task. Thus, we can increase the efficiency of DVS schemes and further reduce the power consumption of the system.

DVS schemes can execute a task set at a lower frequency provided that a schedulability test deems the task set feasible and tasks do not exceed their WCET. For DVS schemes based on earliest-deadline-first (EDF) scheduling, the schedulability test expressed in Equation 2 must be satisfied by the task set to ensure feasibility. Equation 2 represents the utilization of the system under frequency scaling.

$$\frac{C_1}{P_1} + \frac{C_2}{P_2} + \dots + \frac{C_n}{P_n} \leq \alpha \quad (2)$$

$C_1, C_2, \dots, C_n$  represent the WCET for each of the  $n$  tasks.  $P_1, P_2, \dots, P_n$  represent the respective periods of the tasks. As is common in base EDF, tasks’ deadlines are assumed to be equal to their periods. The term  $\alpha$  in Equ-

tion 2 is the scaling factor. It identifies actual (scaled) frequency such that  $\alpha = f_c/f_m$ , where  $f_c$  is the scaled frequency and  $f_m$  is the peak frequency.

Equation 2 does not consider the effect of frequency scaling on WCET. By combining Equation 1 with Equation 2, we yield a more accurate scaling factor taking the effects of frequency scaling on WCET into account, as seen in Equation 3. 
$$\frac{i_1 + \alpha m_1 L f_m}{P_1 f_m} + \dots + \frac{i_n + \alpha m_n L f_m}{P_n f_m} \leq \alpha \quad (3)$$

To obtain Equation 3, we substitute the WCET by  $C_1 = (i_1 + m_1 L f)/f$  derived from Equation 1. The number of cycles,  $N$ , is expressed in terms of the frequency,  $f$ , and the memory latency,  $L$ , using the relation  $N = L \times f$ . Since the schedulability test in Equation 3 is used for frequency scaling, substitute  $f = \alpha f_m$  in WCET equations. The schedulability test in Equation 3 is transformed into Equation 4 to consider the scaling factor.

$$\frac{\sum_{j=1}^n i_j / P_j}{f_m (1 - L \sum_{j=1}^n m_j / P_j)} \leq \alpha \quad (4)$$

The scaling factor in Equation 4 results in a much lower frequency  $f_c$ . The WCET used is not exaggerated, and slack is exploited efficiently.

In our implementation work, we integrated FAST equations into DVS-EDF scheduling as proposed by Pillai and Shin through (a) static voltage scaling, (b) cycle-conserving RT-DVS and (c) look-ahead RT-DVS [19]. With only minimal changes to the original algorithms, we integrated the FAST equations into the respective DVS schemes, thereby improving energy savings obtained.

#### 4.1. FAST - Static Voltage Scaling

The static voltage scaling schemes introduced of Pillai and Shin [19] uses the modified EDF test shown in Equation 2 to calculate the scaling factor  $\alpha$ . This algorithm uses all static slack in the system. The processor frequency for the entire task set is set statically. Dynamic slack produced during runtime due to early completion of tasks is not considered for frequency scaling. The FAST equations for the WCET can be integrated into the static voltage scheme as shown in Figure 8. Equation 1 represents the WCET of all tasks, and the scaling factor is calculated using Equation 4. The FAST static voltage scaling algorithm performs better than the original static voltage scheme because it considers the portion of WCET that scales with frequency.

```
EDF-test( $\alpha$ ):
  if  $\frac{\sum_{j=1}^n i_j / P_j}{f_m (1 - L \sum_{j=1}^n m_j / P_j)} \leq \alpha$  return true ;
  else return false;
select-frequency:
  use lowest frequency
   $f_k \in \{f_1, \dots, f_m \mid f_1 < \dots < f_{max}\}$ 
  such that EDF-test( $f_k / f_{max}$ ) is true;
```

**Figure 8. FAST-Static Voltage Scaling for EDF**

#### 4.2. FAST - Cycle-Conserving RT-DVS

The cycle conserving RT-DVS by Pillai and Shin [19] calculates the utilization for a task set at every task release and task completion. Upon task release, the utilization is calculated based on the WCET. Upon task completion, the utilization is calculated by considering the actual execution time of the completed task instead of the WCET. This algorithm uses the static slack available in the system as well as the dynamic slack generated due to early task completions. Figure 9 shows the necessary modifications to the original algorithm to incorporate the FAST equations.

The FAST cycle conserving DVS scheme outperforms the original scheme since it takes the actual execution times as well the scaling levels of previous tasks into account. The scheme derives the current system utilization after task completion by considering the actual execution time. In FAST cycles-conserving RT-DVS, the total number of cycles and the total number of misses experienced by a task are determined during executing, *e.g.*, by hardware counters, which have become quite common for modern architectures. The actual execution time is also converted into a FAST equation to consider its scaling with frequency. The system utilization and the scaling factor are calculated through Equations 3 and 4.

```
select-frequency():
  use lowest frequency
   $f_k \in \{f_1, \dots, f_{max} \mid f_1 < \dots < f_{max}\}$ 
  such that  $\frac{\sum_{j=1}^n i_j / P_j}{f_m (1 - L \sum_{j=1}^n m_j / P_j)} \leq f_k / f_{max}$  ;
upon task-release( $T_j$ ):
  set  $i_j = i_{WCET}$  and  $m_j = m_{WCET}$  ;
  select frequency();
upon task-completion( $T_j$ ):
  set  $i_j = i_{actual}$  and  $m_j = m_{actual}$  ;
  /* $m_{actual}$  are the actual number of misses
  for this invocation,
   $i_{actual}$  are the ideal number of cycles for
  this invocation not counting the miss cycles*/
  select frequency();
```

**Figure 9. FAST-Cycle conserving DVS for EDF**

#### 4.3. FAST - Look-Ahead RT-DVS

The look-ahead RT-DVS schemes by Pillai and Shin [19] finds the minimum amount of work that may be performed between now and the next scheduling event without missing any deadlines. All work is deferred till the last possible moment, also referred to as last-chance scheduling [8]. As a side effect, the frequency may be increased as execution approaches a deadline. In practice, most tasks complete execution early, *i.e.*, prior to their WCET. Hence, the frequency rarely has to be raised to complete by a deadline. This algorithm also uses all the static slack (idle) as well as most of the dynamic slack. Figure 4.3 depicts the modified original algorithm to integrate the FAST equations into the DVS scheme. Figure 4.3 also shows a modification to the look-

ahead RT-DVS algorithm for *task – completion* by setting  $c\_left_j = C_j$  (see appendix). The FAST look ahead scheme also takes advantage of FAST equations to lower energy consumption of the algorithm. The terms  $i\_left$  and  $m\_left$  describe the computation left in the form of a FAST equation. Hardware counters are employed to track total cycles completed and total misses inflicted while a task is executing. The  $s$  component shown in Figure 4.3 cannot be converted into a FAST equation unless considerable changes are made to the algorithm. Doing so would make the algorithm more aggressive leading to lower frequencies. To avoid excessive modifications, only the next scheduled task is expressed in the form of a FAST equation. The experiments show that the performance of the algorithm is improved even with minimal modifications to the algorithms.

## 5. Experimental Framework

The experimental framework is divided into two sections. The first section is devoted to comparing the WCEC calculated using FAST equations, obtained from the FAST framework, to the WCEC obtained from the traditional static timing analysis tool. The second section tests and compares FAST-DVS algorithms with the original DVS algorithms proposed by Pillai and Shin [19].

### 5.1. Testing the FAST Framework

We re-designed our static timing analyzer [11] to create the FAST framework. The FAST tool, like its predecessor

```

select-frequency(x):    use lowest frequency
                       $f_k \in \{f_1, \dots, f_{max} | f_1 < \dots < f_{max}\}$ 
                      such that  $x \leq f_k / f_{max}$  ;
upon task-release( $T_j$ ):
  set  $c\_left_j = C_j$ ,
     $i\_left_j = i\_wctet_j$  and  $m\_left_j = m\_wctet_j$  ;
  defer();
upon task-completion( $T_j$ ):
  set  $c\_left_j = C_j$ ,
     $i\_left_j = i\_wctet_j$  and  $m\_left_j = m\_wctet_j$  ;
  defer();
during task-execution( $T_j$ ):
  decrement  $c\_left_j$ ,  $i\_left_j$  and  $m\_left_j$  ;
defer():
  set  $U = C_1/P_1 + \dots + C_n/P_n$ ;
  set  $s = 0$  ;
  for  $i = 1$  to  $n$ ,  $T_j \in T_1, \dots, T_n | D_1 \geq \dots \geq D_n$ 
    /*Note: reverse EDF order of tasks*/
    set  $U = U - C_j/P_j$  ;
    set  $x_j = \max(0, c\_left_j - (1 - U)(D_j - D_n))$  ;
    set  $U = U + (c\_left_j - x_j)/(D_j - D_n)$  ;
    set  $s = s + x_j$ ;
 $s = s - x_n$ 
 $t = D_n - \text{current\_time}$ 
 $x = \frac{(i\_left_n + s)/t}{f_m(1 - L \times m\_left_n/t)}$  ;
select-frequency(x);

```

Figure 10. FAST-Look ahead DVS for EDF

[1], is based on the portable ISA (PISA) used by the SimpleScalar tool set. All instruction execution latencies are based on the MIPS R10K latencies. Specifically, a constant memory latency of 100ns is used. We use a 8KB direct-mapped instruction cache and a 8KB direct-mapped data cache. For the instruction cache categorizations, the static cache simulator of our existing tool set is used. To obtain data cache categorizations distinguishing hits and misses, we use a scheme that assumes a constant number of data accesses as misses and the remaining references as cache hits. During pipeline simulation, a static branch prediction scheme using the Ball-Larus heuristic is modeled. Both the static timing analysis tool and the FAST tool model a simple in-order six-stage pipeline.

When incorporating the frequency model into the static timing analyzer, two paths with FAST equations that result in intersecting first-order polynomials may be encountered. In this case, we resort to the third method introduced in Section 3.2 to choose the equation resulting in the worst-case behavior. First, we try to determine if one equation is always greater than the other for the valid range of frequencies (100MHz-1GHz). Otherwise, we approximate the two equations by an equation providing a safe upper bound. This may result in slight overestimations but, overall, still provides sufficiently tight bound of the WCEC, as will be seen. We also remove the branch misprediction penalty from the FAST equation if branch misprediction overlaps with a data miss stall. The overestimation caused by instructions with execution latencies higher than one are not removed from the equation as they contribute insignificant savings.

We studied six real-time benchmarks from the C-lab real-time benchmark suite [5], commonly utilized for WCET experiments. Three floating point benchmarks, adpcm, lms and fft as well as three integer benchmarks, cnt, srt and mm are analyzed. These benchmarks were compiled by the PISA GCC compiler integrated with our SimpleScalar-based tool set. From the compilation of these benchmarks, the control-flow graphs and instruction layouts were obtained, which are taken as inputs to the FAST analyzer and the static cache analyzer. The FAST output is the WCEC in the form of a parametric equation conforming with our parametric frequency model. The same benchmarks were also exposed to the original static timing analysis tool set for comparison. The original static timing analyzer must be run separately for each frequency under consideration to account for changed memory latency for a given processor frequency. In contrast, the FAST framework captures the same effect in an equation (derived from a single analysis step).

### 5.2. Testing FAST-DVS Schemes

To test the FAST-DVS schemes, we implemented the algorithms in a scheduling simulator. Implementation features include generic static voltage scaling support and scheduling algorithms ranging from base EDF, cycle-

conserving RT-DVS, look-ahead RT-DVS, FAST static voltage scaling, FAST cycle conserving RT-DVS to FAST look ahead RT-DVS. All the scheduling algorithms can choose a frequency between 100MHz to 1GHz for the next scheduled task. The base EDF algorithm runs all tasks at 1GHz. All algorithms switch the processor frequency to 100MHz during idle times in the schedule, the lowest available frequency, since it is not realistic to put a processor into sleep mode (with millisecond overheads) for frequent task releases (in the order of milliseconds). A combination of task sets resulting from application workloads of six real-time benchmarks, namely srt, fft, mm, lms, adpcm and cnt, were studied. The task sets were exposed to the simulator, and energy consumption was calculated for all scheduling algorithms. The execution times were derived from exposing the benchmarks to a cycle-accurate pipeline model implemented in our SimpleScalar-based simulator [20]. By exploiting a cycle-accurate architectural simulator, we can obtain the total number of cache misses as well as the total number of cycles executed. The execution times obtained from the architectural simulator are scaled with frequency using the same assumption used while formulating the FAST parametric model. Namely, we assume that the total number of execution cycles does not remain constant with frequency. The same execution time scaling method is used for all the voltage scaling algorithms.

To evaluate the different FAST-DVS and DVS schemes, we formed several tasksets using the cnt, srt, mm, adpcm, fft and lms benchmarks. Three groups were formed as follows - G1: cnt, srt, mm (all integer), G2:adpcm, fft, lms (all floating point) and G3:cnt, mm, fft, lms (mixed). The periods were chosen for each benchmark and from each group two tasksets are created – one with high utilization, and one with low utilization. The high utilization tasksets have a utilization of approximately 0.9 while the low utilization tasksets have a utilization of approximately 0.5.

The frequency/voltage settings used for the scheduling simulator 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 /

Benchmarks	Equations		WCET:Static timing analysis/ FAST (WCEC)			
	i	m	100MHZ	400MHZ	700MHZ	1000MHZ
fft	355933	24658	600628/ 602675	1340578/ 1342625	2079876/ 2081993	2820478/ 2822525
adpcm	3026370	544104	8433905/ 8467410	24749525/ 24790530	41065145/ 41113650	57380765/ 57436770
lms	167890	29905	466438/ 466940	1363598/ 1364090	2260748/ 2261240	3157898/ 3158390
cnt	71221	6066	131880/ 131881	313860/ 313861	495840/ 495841	677820/ 677821
mm	2038538	59134	2629877/ 2629878	4403897/ 4403898	6177917/ 6177918	7951937/ 7951938
srt	3509420	102145	4530868/ 4530870	7595218/ 7595220	10659568/ 10659570	13723918/ 13723920

Table 2. WCEC of FAST vs. Traditional

1.8 V in 25 MHz / 0.03 V increments. We calculate energy per cycle at a particular frequency by using the relation  $Energy = Voltage^2 \times frequency$ .

## 6. Results for FAST Framework

The WCEC equations for the six benchmarks obtained from the static timing analysis tool and the FAST tool are compiled in Table 2 and in Figure 12. The FAST scheme differs from conventional static timing analysis without parametric expressions of frequencies by less than half a percent. Hence, we conclude that the FAST equations accurately model the WCEC obtained from the static analysis tool. Since the effects of scaling on WCEC are accurately modeled by the FAST equations, the scaling of the WCET can also be accurately captured.

Table 2 shows the WCEC for all six benchmarks calculated for four different frequencies using the FAST equations and compared with the corresponding WCEC obtained from the static timing analysis tool. Figure 12 plots the ratio of the WCET for the FAST tool and the static timing analysis tool. As shown in the Table 2 and Figure 12, cnt, mm and srt show that the FAST bounds on WCET match the bounds obtained by the static timing analyzer exactly. For fft, adpcm and lms the FAST bounds on WCET are very close to the bounds obtained by the static timing analyzer. The overestimation in these benchmarks is due to the presence of floating point operations that have overlapping execution latencies with memory stalls (see Section 2.2, Figure 5). Thus, the FAST tool can accurately model the WCEC of tasks with a negligible error (<1%) by using our parametric frequency model.

## 7. Results for FAST-DVS Schemes

Figures 11(a) to 11(f) show the Energy for all the DVS schemes normalized to the base EDF scheme for all six tasksets. The figures show a decrease in power consumption

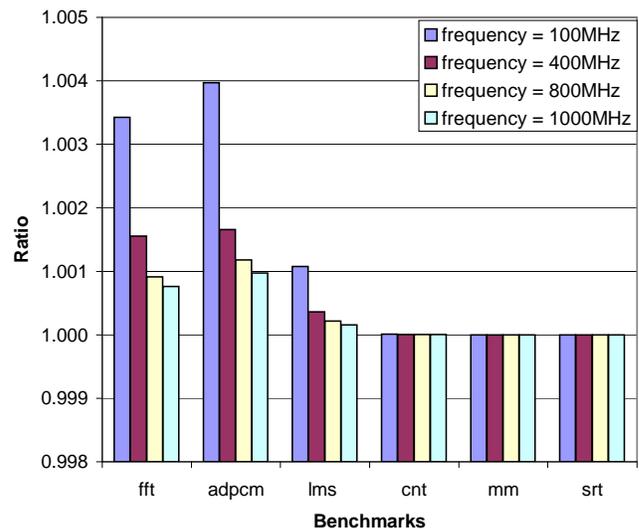
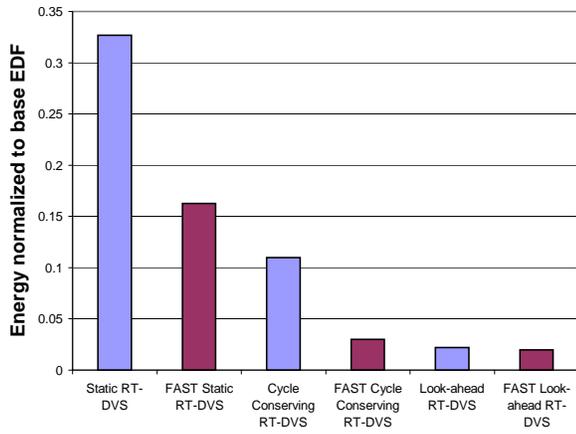
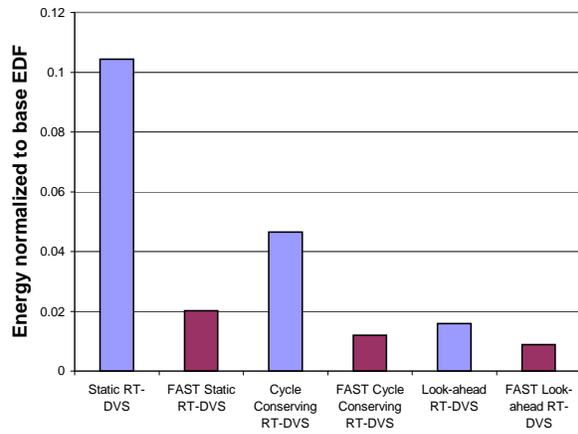


Figure 12. FAST vs. Traditional WCEC



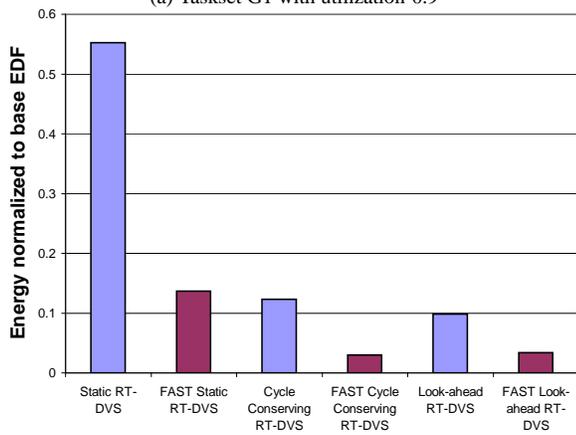
DVS-EDF Schemes

(a) Taskset G1 with utilization 0.9



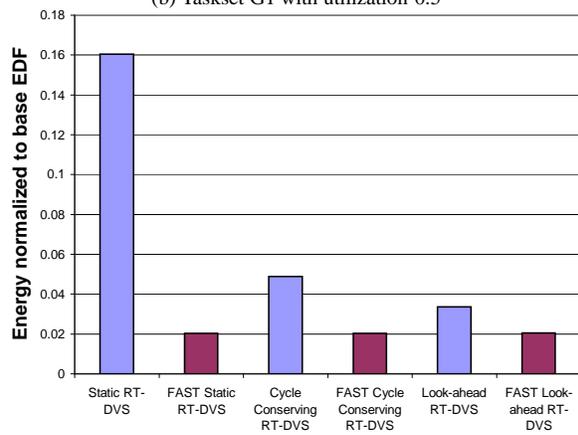
DVS-EDF Schemes

(b) Taskset G1 with utilization 0.5



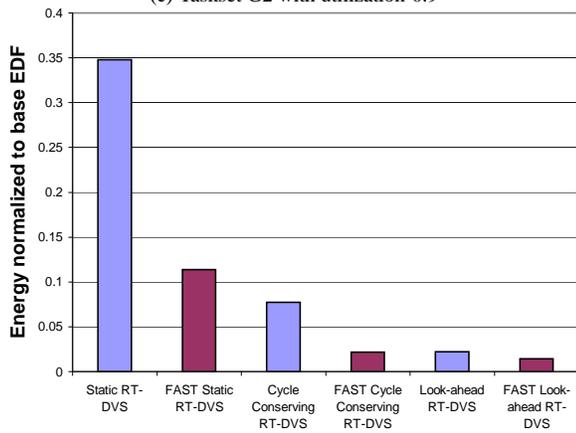
DVS-EDF Schemes

(c) Taskset G2 with utilization 0.9



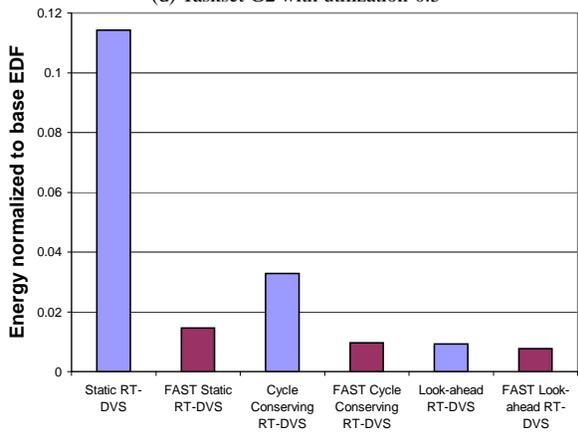
DVS-EDF Schemes

(d) Taskset G2 with utilization 0.5



DVS-EDF Schemes

(e) Taskset G3 with utilization 0.9



DVS-EDF Schemes

(f) Taskset G3 with utilization 0.5

**Figure 11. Energy Normalized to Base EDF for Various Task Sets**

tion for all the FAST-DVS schemes when compared to the original RT-DVS schemes. The first, third and fifth bars in the graphs show the energy consumption for the original RT-DVS schemes. The second, fourth and sixth bars in the graphs show the improved energy consumption for the FAST-DVS schemes.

For the integer taskset G1, savings are considerable (ex-

cess of 50%) between the original scheme and the corresponding FAST scheme for the static and cycle-conserving approaches (Figures 11(a) and 11(b)). The look-ahead scheme shows none or only marginal savings under FAST for high and lower utilizations, respectively. This is caused by fact that the FAST look-ahead scheme runs the taskset at a lower frequency and has to recover by raising the fre-

quency more often than the original look-ahead scheme.

The results are also sensitive to the task set, as a comparison with the floating-point taskset G2 shows. Figures 11(c) and 11(d) indicate that G2 still experiences considerable savings for high utilizations – and slightly lower ones for lower utilizations – under the corresponding FAST scheme. In case of G2, savings for the static and cycle-conserving schemes are even higher. The results for the integer/floating point mix of G3 in Figures 11(e) and 11(f) show savings at levels between the G1 and G2 tasksets for static and cycle-conserving schemes. The look-ahead version of FAST results in less significant savings, mostly due to already very aggressive savings due to the original look-ahead scheme.

All results depend on the FAST equation for the benchmarks. The scalability of the WCET depends on the number of misses counted during timing analysis. Due to a worst-case analysis, the number of misses are usually highly exaggerated, especially for data caches. This means that the original schemes are penalized heavily due to their assumptions about scaling the WCET. Using the FAST equations, the DVS schemes can improve the tightness of the WCET, which is already highly exaggerated, thereby improving energy consumption.

Overall, FAST equations with the RT-DVS schemes are more greedy and results in lower frequencies. The relative energy benefits are highest in the static RT-DVS scheme because it has the most scope for improvement. The cycle conserving and the look-ahead RT-DVS schemes are dynamic schemes and already scale the frequency aggressively. The addition of the FAST equations to these aggressive schemes enables them to scale the frequency even more aggressively, showing lower energy consumption. Hence, benefits for FAST are being observed in all cases.

## 8. Related Work

Recently, a number of research groups have addressed various issues in the area of predicting the worst-case execution time (WCET) of real-time programs. Conventional methods for static analysis have been extended from unoptimized programs on simple CISC processors to optimized programs on pipelined RISC processors, and from uncached architectures to instruction and data caches [18, 14, 12, 17, 23, 13]. All these methods obtain discrete values to bound the WCET in a non-parametric fashion.

Vivancos *et al.* describe techniques for addressing static timing analysis for variable loop bounds [21]. The so-called parametric timing analysis allows dynamic schedulers to reassess the WCET based on dynamically determined loop bounds during program execution. Chapman *et al.* [7] used path expressions to combine a source-oriented parametric approach of WCET analysis with timing annotations, verifying the latter through the former. Bernat and Burns also proposed using algebraic expressions to represent the

WCET of subprograms, where the algebraic expression is parameterized by some of the subprogram's parameters [4]. These approaches differ in that they address fundamental problems in static timing analysis. Our FAST approach, in contrast, aims at isolating execution effects as a function of the processor frequency, a unique, unprecedented approach complementing existing work on static timing analysis.

## 9. Conclusion

In this work, novel techniques for tight and flexible static timing analysis were developed most suitable – but not restricted to – dynamic scheduling schemes. The essence of our approach lies in providing frequency-aware bounds on the WCET through static timing analysis. Using a frequency-sensitive parametric model, we can capture the effect of combined DFS/DVS on the WCEC and, thus, accurately model the WCET over any frequency range. These techniques are implemented in a frequency-aware static timing analysis (FAST) tool leveraging prior expertise on static timing analysis. Experiments show the capability of FAST to derive safe upper bounds on the WCET, which are almost as tight (within 1%) as conventional, non-parametric timing analysis. FAST equations can also be used to improve existing DVS scheduling schemes to ensure that the effect of frequency scaling on WCET is considered and that the WCET used is not exaggerated. This is demonstrated by incorporating FAST into three DVS scheduling schemes. Results indicate significant energy savings over the base DVS schedulers due to FAST. To the best of our knowledge, this study of DVS effects on timing analysis is unprecedented.

## Acknowledgments

The improvements of Look-ahead DVS-EDF were designed together with Harini Ramaprasad and Sibin Mohan.

## References

- [1] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (VISA): Exceeding the complexity limit in safe real-time systems. In *International Symposium on Computer Architecture*, pages 250–261, June 2003.
- [2] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 2001.
- [3] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, June 1988.
- [4] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *25th IFAC Workshop on Real-Time Programming*, May 2000.
- [5] C-Lab. Wcet benchmarks. Available from <http://www.c-lab.de/home/en/download.html>.

- [6] A. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power cmos digital design. In *IEEE Journal of Solid-State Circuits*, Vol. 27, pp. 473-484., April, 1992.
- [7] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145-171, 1996.
- [8] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261-1269, Oct. 1989.
- [9] A. Dudani, F. Mueller, and Y. Zhu. Energy-conserving feedback edf scheduling for embedded systems with real-time constraints. In *ACM SIGPLAN Joint Conference Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPE5'02)*, pages 213-222, June 2002.
- [10] F. Gruian. Hard real-time scheduling for low energy using stochastic data and dvs processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01*, Aug 2001.
- [11] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53-70, Jan. 1999.
- [12] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288-297, Dec. 1995.
- [13] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254-263, Dec. 1996.
- [14] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97-108, Dec. 1994.
- [15] 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.
- [16] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low Power*, Oct. 2000.
- [17] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209-239, May 2000.
- [18] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31-61, Mar. 1993.
- [19] P. Pillai and K. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Symposium on Operating Systems Principles*, 2001.
- [20] E. Rotenberg. Using variable-Mhz microprocessors to efficiently handle uncertainty in real-time systems. In *34th International Symposium on Microarchitecture*, pages 28 - 39, Dec. 2001.
- [21] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Embedded Systems*, Aug. 2001.
- [22] 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.
- [23] 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.

## Modified Look-ahead DVS-EDF

A number of DVS schemes were proposed by Pillai and Shin for scheduling hard real-time systems [19]. A simple, *static* scaling version uniformly scales the frequency for all tasks based on utilization tests for schedulability, both for rate-monotone and EDF scheduling. *Cycle-conserving* EDF lowers utilization upon task completion temporarily to the proportion of the actual execution time. *Look-ahead* EDF is an extension to these scheme that capitalizes on early task completion by deferring work for future tasks in favor of scaling the current task. Scaling of the current task occurs based on a modified utilization test that benefits from both idle slots and early task completion. At any completion (both early and on time), the utilization is effectively reduced for the completing task (up until its next release time).

Specifically, upon task completion,  $cc_i = c_{left}_1 = 0$  according to Cycle-Conserving EDF and Look-ahead EDF, respectively. The *defer* calculations of Look-ahead EDF then reassesses the utilization based on future and past deadlines for released and completed tasks, respectively.

We modified the Look-ahead EDF by setting  $c_{left}_i = C_i$  at task completion instead of assigning a zero value. In addition, we reassess the utilization *strictly* based on the next deadline in the future, irregardless of whether tasks are already released and not. This allows us to look ahead even further in the schedule and, thereby, potentially save additional energy by lowering frequencies more aggressively, and it retains the safety of the schedule by adhering to the EDF utilization test. If the WCET is not fully utilized, then other tasks may still benefit from early completion up to the threshold given by the idle times left in the schedule. This modified Look-ahead EDF scheme was implemented in our comparison and is shown to result in up to 34% lower energy consumption than the original scheme. On the average, the modified scheme saves an additional 5-11% of energy for utilizations between 25% and 100%. At high utilizations, our modification occasionally requires between 0.5-8% more energy, which is due to considering an actual time of  $cc_i = 0$  in the original scheme up to the next release of a task. Hence, it would be possible to switch between the two schemes based on a utilization threshold as a trigger. Additional savings over the modified scheme due to early completion can only be obtained by considering the density of a schedule at some instance in time, such as given by the maximal schedule utilized in our feedback EDF scheme.