Parametric Timing Analysis and Its Application to Dynamic Voltage Scaling

SIBIN MOHAN, FRANK MUELLER, North Carolina State University
MICHAEL ROOT, WILLIAM HAWKINS, CHRISTOPHER HEALY, Furman University
DAVID WHALLEY, Florida State University
and EMILIO VIVANCOS, Universidad Politecnica de Valencia

Embedded systems with real-time constraints depend on a-priori knowledge of worst-case execution times (WCETs) to determine if tasks meet deadlines. Static timing analysis derives bounds on WCETs but requires statically known loop bounds.

This work removes the constraint on known loop bounds through parametric analysis expressing WCETs as functions. Tighter WCETs are dynamically discovered to exploit slack by dynamic voltage scaling (DVS) saving 60%-82% energy over DVS-oblivious techniques and showing savings close to more costly dynamic-priority DVS algorithms.

Overall, parametric analysis expands the class of real-time applications to programs with loop-invariant dynamic loop bounds while retaining tight WCET bounds.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*scheduling*; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Real-Time Systems, Worst-Case Execution Time, Timing Analysis, Dynamic Voltage Scaling

This work was conducted at North Carolina State University and Florida State University; it was supported in part by NSF grants CCR-0208581, CCR-0310860, CCR-0312695, EIA-0072043, CCR-0208892, CCR-0312493 and CCR-0312531.
Author's address: Sibin Mohan, Frank Mueller, Dept. of Computer Science, Center for Embedded Systems Research, North Carolina State University, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu, +1.919.515.7889
Christopher Healy, Michael Root, William Hawkins, Dept. of Computer Science, Furman University, Greenville, SC 29613, chris.healy@furman.edu
David Whalley, Dept. of Computer Science, Florida State University, Tallahassee, FL 32306, whalley@cs.fsu.edu
Emilio Vivancos, Department de Sistemas Informaticos y Computacion, Universidad Politecnica de Valencia, 46022-Valencia, Spain, vivancos@dsic.upv.es
Preliminary versions of this material appeared in LCTES'01 [Vivancos et al. 2001] and RTSS'05 [Mohan et al. 2005].
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 1539-9087/20YY/0200-0001 $5.00

1. INTRODUCTION

Real-time and embedded systems are increasingly deployed in safety-critical environments. Examples include avionics, power plants, automobiles, *etc.* The software, in general, must be validated, which traditionally amounts to checking the correctness of the input/output relation. Many embedded systems also impose timing constraints, which, if violated, may not only render a system non-functional, but may also result in fallouts dangerous to the environment. Such systems are commonly referred to as real-time systems, and they impose timing constraints (termed as deadlines) on computational tasks to ensure that results are provided on time. Often, approximate results supplied on time are preferred to more precise results that may become available late, *i.e.*, after the deadlines have passed. One critical piece of information required by designers of real-time systems to verify that tasks meet their deadlines, is the worst-case execution time (WCET) of each task. Bounds on WCETs of tasks are automatically determined by static timing analysis tools. The total time in the schedule and each task's WCET can subsequently be used to make scheduling decisions.

Static timing analysis [Puschner and Koza 1989; Harmon et al. 1992; Park 1993; Lim et al. 1994; Healy et al. 1995; Chapman et al. 1996; Li et al. 1996; Malik et al. 1997; Healy et al. 1998; White et al. 1999; Mueller 2000; Hergenhan and Rosenstiel 2000; Bernat and Burns 2000; Wegener and Mueller 2001; Chen et al. 2001; Engblom et al. 2001; Engblom 2002; Bernat et al. 2002; Thesing et al. 2003; Mohan et al. 2005] provides bounds on the WCET. The *tighter* these bounds relative to the true worst-case times, the greater the value of the analysis. Of course, even a tight bound has to be a *safe bound* in that it must not underestimate the true WCET; it may only match it or exceed it. In general, timing analysis is by no means an easy or trivial task. Bounds on execution times require constraints to be imposed on the tasks (timed code), the most striking of which is the requirement to statically bound the number of iterations of loops within the task. These loop bounds address the halting problem, *i.e.*, without these loop bounds, WCET bounds cannot be derived. The programmer must provide these upper bounds on loop iterations when they cannot be inferred by program analysis. Hence, these statically fixed loop bounds may present an inconvenience. They also restrict the class of programs that can be used in real-time systems. This type of timing analysis is referred to as *numeric* timing analysis [Harmon et al. 1992; Healy et al. 1995; White et al. 1997; Healy et al. 1998; White et al. 1999; Mueller 2000] since it results in a single numeric value for WCET given the upper bounds on loop iterations.

The constraint on the known maximum number of loop iterations is removed by *parametric* timing analysis (PTA) [Vivancos et al. 2001]. PTA permits variable length loops. Loops may be bounded by n iterations as long as n is known prior to loop entry during execution. Such a relaxation widens the scope of analyzable programs considerably and facilitates code reuse for embedded/real-time applications.

This paper derives (a) parametric expressions to bound WCET values of dynamically bounded loops as polynomial functions. The variables affecting execution time, such as a loop bound n, constitute the formal parameters of such functions, while the actual value of n at execution time is used to evaluate such a function. This paper further describes (b) the application of static timing analysis techniques to dynamic scheduling problems and (c) assesses the benefits of PTA for dynamic voltage scaling (DVS). This work contributes a novel technique that allows PTA to interact with a dynamic scheduler while

discovering actual loop bounds, during execution, prior to loop entry. At loop entry, a tighter bound on WCET can be calculated on-the-fly, which may then trigger scheduling decisions synchronous with the execution of the task. The benefits of PTA resulting from this dynamically discovered slack are analyzed. This slack could be utilized in two ways – *(a)* execution of additional tasks as a result of admissions scheduling, and *(b)* power management.

Recently, numerous approaches have been presented that utilize DVS for both, general-purpose systems [Weiser et al. 1994; Govil et al. 1995; Pering et al. 1995; Grunwald et al. 2000] and for real-time systems [Gruian 2001; Shin et al. 2000; Pillai and Shin 2001; Aydin et al. 2001; Shin et al. 2001; Aydin et al. 2001; Kang et al. 2002; Zhang and Chanson 2002; Saewong and Rajkumar 2003; Lee and Krishna 2003; Liu and Mok 2003]. Core voltages of contemporary processors can be reduced while lowering execution frequencies. At these lower execution rates, power is significantly reduced, as power is proportional to the frequency and to the square of the voltage: $P \propto V^2 \times f$.

In the past, real-time scheduling algorithms have shown how static and dynamic slack may be exploited in inter-task DVS approaches [Gruian 2001; Shin et al. 2000; Pillai and Shin 2001; Aydin et al. 2001; Kang et al. 2002; Zhang and Chanson 2002; Saewong and Rajkumar 2003; Lee and Krishna 2003; Liu and Mok 2003; Lee and Shin 2004; Zhu and Mueller 2004; 2005; Jejurikar and Gupta 2005; Zhong and Xu 2005] as well as intra-task DVS algorithms [Mosse et al. 2000; Shin et al. 2001; Aydin et al. 2001; AbouGhazaleh et al. 2001]. Early task completion and techniques to assess the progress of execution based on past executions of a task lead to dynamic slack discovery.

We use a novel approach towards dynamic slack discovery. Slack, in our method, can be *safely predicted for future execution* by exploiting early knowledge of parametric loop bounds. This allows us to tightly bound the remainder of execution of a task. The potential for dynamic power conservation via *ParaScale*, a novel intra-task DVS algorithm, is assessed. ParaScale allows tasks to be *slowed down* as and when more slack becomes available. This is in sharp contrast to past real-time DVS schemes, where tasks are sped up in later stages as they approach their deadline [Gruian 2001; Lee and Shin 2004; Zhu and Mueller 2004; 2005; Jejurikar and Gupta 2005; Zhong and Xu 2005].

We also implemented a novel enhancement to the static DVS scheme and incorporated it with our intra-task slack determination scheme resulting in significant energy savings. The energy savings approach those obtained by one of the most aggressive dynamic DVS algorithms [Pillai and Shin 2001].

The approach is evaluated by implementing PTA in a gcc environment with a MIPS-like instruction set. Execution is simulated on a customized SimpleScalar [Burger et al. 1996] framework that supports multi-tasking. We bound the effect of instruction cache misses but not data cache misses in our experiments. The framework has been modified to support customized schedulers with and without DVS policies and an enhanced Wattch power model [Brooks et al. 2000], which aids in assessing power consumption. We also implemented a more accurate leakage power model similar to [Jejurikar et al. 2004] to estimate the amount of leakage and static power consumed by the processor. This framework is used to study the benefits of PTA in the context of ParaScale as a means to exploit DVS.

Our results indicate that ParaScale, applied on a modified version of a static DVS algorithm, provides significant savings by utilizing our parametric approach to timing analysis. These savings are observed for generated dynamic slack and potential reduction in overall

energy. In fact, the amount of energy saved is very close to that obtained by the lookahead EDF-DVS scheme [Pillai and Shin 2001] – a popular, aggressive *dynamic* DVS algorithm. Thus, ParaScale makes it possible for static inter-task DVS algorithms to be used on embedded systems. This helps avoid more cumbersome (and difficult to implement) DVS schemes while still achieving similar energy savings. Our approach utilizes online intra-task DVS to exploit parametric execution times resulting in much lower power consumptions, *i.e.*, even without any scheduler-assisted DVS savings. Hence, even in the absence of dynamic priority scheduling, significant power savings may be achieved, *e.g.*, in the case of cyclic executives or fixed-priority policies such as rate-monotonic schedulers [Liu and Layland 1973]. Overall, parametric timing analysis expands the class of applications for real-time systems to include programs with dynamic loop bounds that are loop invariant while retaining tight WCET bounds and uncovering additional slack in the schedule.

The paper is structured as follows. Sections 2 and 3 provide information on numeric as well as parametric timing analysis. Section 4 explains derivation of the parametric formulae and their integration into the code of tasks. This section also shows the steps involved in obtaining accurate WCET analysis for the new, enhanced code. Section 5 discusses the context in which parametric timing results are used. Section 6 introduces the simulation framework. Section 7 elaborates on the experiments and results. Section 8 discusses related work, and Section 9 summarizes the work.

2. NUMERIC TIMING ANALYSIS

Knowledge of worst-case execution times (WCETs) is necessary for most hard real-time systems. The WCET must be known or safely bounded *a priori*, so that the feasibility of scheduling task sets in the system may be determined, given a scheduling policy, such as rate-monotonic or earliest-deadline-first scheduling [Liu and Layland 1973]. Timing analysis methods typically fall into two categories – *static* and *dynamic*. It has been shown that dynamic timing analysis methods, based on trace-driven or experimental methods, cannot guarantee the safety of WCET values obtained [Wegener and Mueller 2001]. Architectural complexities, difficulties in determining worst-case input sets and the exponential complexity of performing exhaustive testing over all possible inputs are also reasons why dynamic timing analysis methods are infeasible in general.

In contrast, static timing analysis methods guarantee upper bounds on WCET of tasks. In this work, we constrain ourselves to a toolset developed in our previous work [Healy et al. 1999; Mueller 2000; White et al. 1999; Mohan et al. 2005]. Static timing analysis models the traversal of all possible execution paths in the code. Execution timing is determined independent of program traces or input data to program variables. The behavior of architectural components is captured as execution paths are traversed. Paths are composed to form functions, loops, etc. until finally the entire application is covered. Hence, we obtain a bound on the WCET and the worst-case execution cycles (WCECs).

The organization of this timing analysis framework is presented in Figure 1. An optimizing compiler is modified to produce control-flow and branch constraint information, as a side-effect of the compilation process. Control-flow graphs and instruction and data references are obtained from assembly code. One of the prerequisites of traditional static timing analysis is that an upper bound on the number of loop iterations be provided to the system.

The control-flow information is used by a static instruction cache simulator to con-

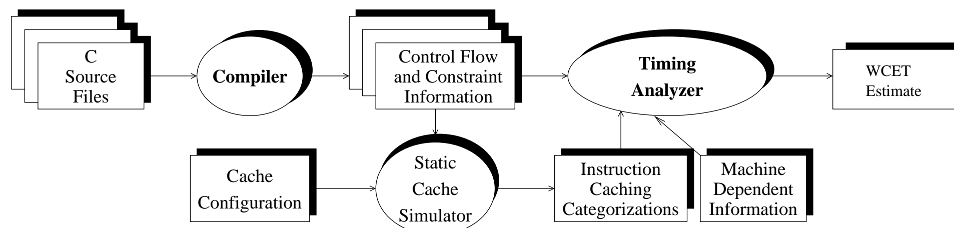


Fig. 1. Static Timing Analysis Framework

struct a control-flow graph of the program and caching categorizations for each instruction [Mueller 2000]. This control-flow graph consists of the call graph and the control flow for each function. The control-flow graph of the program is analyzed, and a caching categorization for each instruction and data reference in the program is produced using a data-flow equation framework. Each loop level containing the instruction and data references is analyzed to obtain separate categorizations. These categorizations for instruction references are described in Table I. Notice that references are conservatively categorized as always-misses if static cache analysis cannot safely infer hits on one or more references of a program line.

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 I. Instruction Categories for WCET

The control-flow, the constraint information, the architecture-specific information and caching categorizations are used by the timing analyzer to derive WCET bounds. Effects of data hazards (load-dependent instruction stalls if a use immediately follows a load instruction), structural hazards (instruction dependencies due to constraints on functional units), and cache misses (obtained from the caching categorizations) are considered by a pipeline simulator for each execution path through a function or loop. We can accommodate static branch prediction in the WCET analysis by adding the misprediction penalty to the non-predicted path.

Path analysis is then performed to select the longest execution path, and once timing results for alternate paths are available, a fixed-point algorithm quickly converges to safely bound the time for all iterations of a loop. Figure 2 illustrates an abstraction of the fix-point algorithm used to perform loop analysis. The algorithm repeatedly selects the longest path through the loop until a fixed point is reached (i.e., the caching behavior does not change and the cycles for the worst-case path remains constant for subsequent loop iterations). WCETs for inner loops are predicted before those for outer loops; an inner loop is treated as a single node for outer loop calculations, and the control flow is partitioned if the number of paths within a loop exceeds a specified limit [Al-Yaqoubi 1997]. The correctness of this fixed-point algorithm has been studied in detail [Arnold et al. 1994].

```
cycles = iter = 0;
do {
   iter = iter + 1;
   wcpath = find the longest path;
   cycles = cycles + wcpath→cycles;
} while (caching behavior of wcpath changes
         && iter < max_iter);
cycles += (wcpath→cycles * (max_iter - iter));
```

Fig. 2. Numeric Loop Analysis Algorithm

By composing the WCET bounds for adjacent paths, the WCET of loops, functions and the entire task is then derived by the timing analyzer by the traversal of a timing tree, which is processed in a bottom up manner. WCETs for outer loop nest/caller functions are not evaluated until the times for inner loop nests/callees are calculated.

3. PARAMETRIC TIMING ANALYSIS

In the static timing analysis method presented above, upper bounds on loop iterations must be known. They can be provided by the user or may be inferred by analysis of the code. This severely restricts the class of applications that can be analyzed by the timing analyzer. We refer to this class of timing analyzers as *numeric timing analyzers* since they provide a single, numeric cycle value provided that upper loop bounds are known.

Parametric timing analysis (PTA) [Vivancos et al. 2001], in contrast, makes it possible to support timing predictions when the number of iterations for a loop is not known until run-time.

Consider the example in Figure 3. The for loop denotes application code traditionally subject to numerical timing analysis for an annotated upper loop bound of 1000 iterations. PTA requires that the value of n be known prior to loop entry. The bold-face code denotes additional code generated by PTA.

```
call IntraTaskScheduler(eval_loop_k(n));
for (i = 0; i<n ; i++ ) // max n = 1000
      loop body ;

// Parametric Evaluation Function
int eval_loop_k(int loop_bound) {
      return (102 * loop_bound);
}
```

Fig. 3. Use of Parametric Timing Analysis

The concept is to calculate a formula (or closed form) for the WCET of a loop, such that the formula depends on *n*, the number of iterations of the loop. The calculation of this formula, [102*n in Figure 3], needs to be relatively inexpensive since it will be used at run-time to make scheduling decisions. These decisions may entail selection/admission of additional tasks or modulation of the processor frequency/voltage to conserve power. Hence, instead of passing a numeric value representing the execution cycles for loops or functions up the timing tree, a symbolic formula is provided if the number of iterations of a loop is not known.

The algorithm in Figure 4 is an abstraction of the revised loop analysis algorithm for PTA. This algorithm iterates to a fixed point, *i.e.*, until the caching behavior does not change. The number of base cycles obtained from this algorithm is then saved. The

```
cycles = iter = 0;
do {
   iter = iter + 1;
   wcpath = find the longest path;
   cycles = cycles + wcpath→cycles;
} while (caching behavior of wcpath changes);
base_cycles = cycles - (wcpath→cycles * iter);
```

Fig. 4. Parametric Loop Analysis Algorithm

$base_cycles$ denote the extra cycles cumulatively inflicted by initial loop iterations before the cycles of the worst-case path reach a fixed point ($wcpath \rightarrow cycles$). The base cycles are subsequently used to calculate the number of cycles in a loop as follows:

$$\mathbf{WCET_{loop}} = \mathbf{wcpath} \rightarrow \mathbf{cycles} * \mathbf{n} + \mathbf{base_cycles} \tag{1}$$

The correctness of this approach follows from the correctness of numeric timing analysis [Healy et al. 1999]. When instruction caches are present in the system, the approach assumes monotonically decreasing WCETs as the cache behavior of different paths through the loop is considered. This integrates well with our past techniques on bounding the worst-case behavior of instruction and data caches [Mueller 2000; White et al. 1999].[1]

Equation 1 illustrates that the WCET of the loop depends on the base cycles and the WCET path time (both constants) as well as on the number of loop iterations, which will only be known at run-time for variable-length loops. The potentially significant savings from such parametric analysis over the numeric approach are illustrated and discussed later in Figure 7. The algorithm in Figure 4 is an enhancement of the algorithm presented in Figure 2. Since the cycles for the worst-case path for the algorithm in Figure 2 has been shown to be monotonically decreasing, the worst-case path cycles for the algorithm in Figure 4 also monotonically decreases.

If the actual number of iterations (say: 100) exceeds the number of iterations required to reach the fixed point for calculating the base cycles (say: 5), then the parametric result closely approximates that calculated by the numeric timing analyzer. If, on the other hand, the actual number of iterations (say: 3) is lower than the fixed point (say: 5), then there could be an overestimation due to considering cycles on top of the WCET path cost (for iterations 4 and 5). The formulae could be modified to deal with the special case that has fewer iterations, *e.g.*, by early termination of our algorithm if actual bounds are lower than the fixed point (future work).

The general constraints on loops that can be analyzed by our parametric timing analyzer are:

(1) Loops must be structured. A structured loop is a loop with a single entry point (*a.k.a* reducible loop) [Aho et al. 1986; Unger and Mueller 2002].

(2) The compiler must be able to generate a symbolic expression to represent the number of loop iterations.

(3) Rectangular loop nests can be handled, as long as the induction variables of these loops are independent of one other.

(4) The value of the *actual* loop bound must be known prior to entry into the loop

[1] Other cache modeling techniques or consideration of timing anomalies due to caches [Berg 2006] may require exhaustive enumeration of all paths and cache effects within the loop or an entirely different algorithm.

$$
\begin{aligned}
&//\, induction_variable && : \ strictly\ monotonically\ increasing/decreasing\ value;\\
&//loop_invariant_variable && : \ loop\ invariant\ relative\ to\ all\ nested\ loops\ up\ to\\
&// && \quad\ outermost\ parametric\ loop\\[1em]
&\mathbf{induction_operation_value} && : \ <\mathbf{constant}> \,||\, <\mathbf{loop_invariant_variable}>\\
&initialization && : \ induction_variable = <induction_operation_value>;\\
&loop && : \ <for, while, do><termination_condition>\\
& && \qquad \#\mathbf{pragma\ max(100)}\\
& && \qquad <body>\\
&body && : \ <statement>;\\
& && \quad\ <induction_variable><op><induction_operation_value>;\\
&op && : \ += \,||\, -=\\
&condition && : \ <induction_variable><comparison_op>\\
& && \qquad <induction_operation_value>
\end{aligned}
$$

Fig. 5. Syntactic and Semantic specifications for constraints on analyzable loops.

Syntactic and semantic specifications that suffice to meet these constraints are presented in Figure 5. The pragma value is the pessimistic worst-case bound for the number of loop iterations. Figure 5 is only informative. Actual analysis is performed on the intermediate code representation. Hence, we are able to handle transformations due to compiler optimizations, *e.g.*, loop unrolling.

The timing analyzer processes inner loops before outer loops, and nested inner loops are represented as single blocks when processing a path in the outer loop. We represent loops with symbolic formulae (rather than a constant number of cycles) when the number of iterations is not statically known. The WCET for the outer loop is simply the symbolic sum of the cycles associated with a formula representing the inner loop as well as the cycles associated with the rest of the path.

The analysis becomes more complicated when paths in a loop contain nested loops with parametric WCET calculations of their own. Consider the example depicted in Figure 6, which contains two loops, where an inner loop (block 4) is nested in the outer loop (blocks 2, 3, 4, 5). Assume that the inner loop is also parametric with a symbolic number of

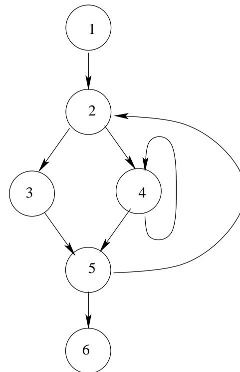


Fig. 6. Example of an outer loop with multiple paths

iterations. The loop analysis algorithm requires that the timing analysis finds the longest path in the outer loop. This obviously depends on the number of iterations of the inner loop. The minimum number of iterations for a loop is one, assuming that the number of loop iterations is the number of times that the loop header (loop entry block) is executed. If the WCET for path A (2→3→5) is less than the WCET for path B (2→4→5), for a single iteration, then path B is chosen, else a *max()* function must be used to represent the parametric WCET of the outer loop. Equation 2 illustrates this idea of calculating the maximum of the two paths. Note though, that the WCET of these paths is obtained after the caching behavior reaches a steady state, and the base cycles are the extra cycles before either of these paths reach that steady state. The first value passed to the *max()* function in this example would be numeric, while the second value would be symbolic.

$$WCET_{loop} = max(WCET_{path_A_time}, WCET_{path_B_time}) * n + base_cycles \quad (2)$$

Similar to numeric timing analysis, certain restrictions still apply. Indirect calls and unstructured loops (loops with more than one entry point) cannot be handled. Recursive functions can, in theory, be handled if the recursion depth is known statically or if the depth can be inferred dynamically prior to the first function call (*via parametric analysis*). Upper bounds on the loop iterations, parametric or not, still need not be known but the bounds can be pessimistic as the actual bounds are now discovered during runtime. In addition, the timing analysis framework has to be enhanced to automatically generate symbolic expressions reflecting the parametric overhead of loops, which will be evaluated at runtime.

Table II shows the results of predicting execution time using the two types of techniques. For these programs we predicted pipeline and instruction cache performance. *Formula* is

Program	Formula	Iters	Observed Cyc.	Numeric Analysis		Param. Analysis	
				Est. Cyc.	Ratio	Est. Cyc.	Ratio
Matcnt	$160n^2 + 267n + 857$	100	1,622,034	1,627,533	1.003	1,627,557	1.003
Matmul	$33n^3 + 310n^2 + 530n + 851$	100	33,725,782	36,153,837	1.072	36,153,851	1.072
Stats	$1049n + 1959$	100	106,340	106,859	1.005	106,859	1.005

Table II. Examples of Parametric Timing Analysis

the formula returned by the parametric timing analyzer and represents the parametrized predicted execution time of the program. In order to evaluate the accuracy of the parametric timing analysis approach, we ensure that each loop in these test programs iterates the same number of times. Thus, n Iters represents the number of loop iterations for each loop in the program and n also represents that value in the formulae. The power of n represents the loop nesting level and the factor represents the cycles spent at that level. Note that most of the programs had multiple loops at each nesting level. For example, $160n^2$ indicates that 160 cycles is the sum of the cycles that would occur in a single iteration of all the loops at nesting level 2 in the program. If the number of iterations of two different loops in a loop nest differ, then the formula would reflect this as a multiplication of these factors. For instance, if the matrix in Matcnt had m rows and n columns, where m≠n, then the formula would be $(160n + 267)m + 857$. Parametric timing analysis supports any rectangular loop nest of independent bounds known prior to loop entry, obtaining bounds for each loop in an inner-most-out fashion using the algorithm in Figure 4. An extension could

handle triangular loops with bounds dependent on outer iterators as well [Healy et al. 2000]. The *Observed Cycles* were obtained by using an integrated pipeline and instruction cache simulator, and represents the cycles of execution given worst-case input data. The *Numeric Analysis* represents the results using the previous version of the timing analyzer, where the number of iterations of each loop is bounded by a number known to the timing analyzer. *Parametric Analysis* represents cycles calculated at run-time when the number of iterations is known and, in this case, equal to the static bound. *Estimated Cycles* and *Ratio* represent the predicted number of cycles by the timing analyzer and its ratio to the Observed Cycles. The estimated parametric cycles were obtained by evaluating the number of iterations with the formula returned by the parametric timing analyzer. These results indicate that the parametric timing analyzer is almost as accurate as the numeric analyzer.

PTA enhances this code with a call to the intra-task scheduler and provides a dynamically calculated, tighter bound on the WCET of the loop. The tighter WCET bound is calculated by an evaluation function generated by the PTA framework. It performs the bounds calculation based on the dynamically discovered loop bound n. The scheduler has access to the WCET bound of the loop derived from the annotated, static loop bound by static timing analysis. It can now anticipate dynamic slack as the difference between the static and the parametric WCET bounds provided by the evaluation function. Without parametric timing analysis, the value of n would have been assumed to be the maximum value, *i.e.*, 100 in this case.

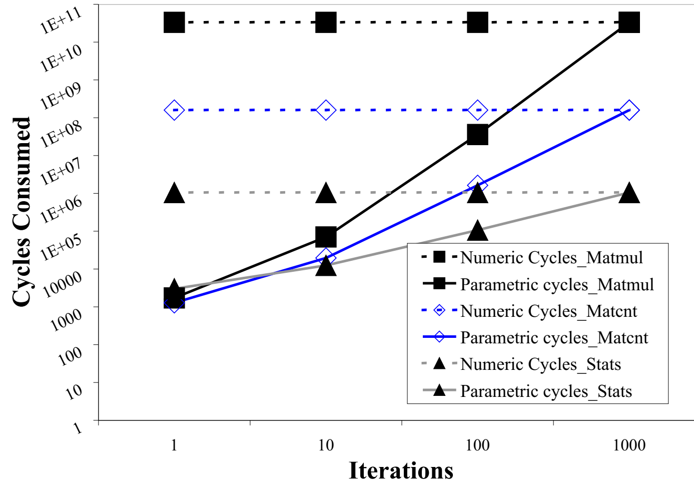


Fig. 7. WCET Bounds as a Function of the Number of Iterations

Figure 7 shows the effect of changing the number of iterations on loop bounds for parametric and numerical WCET analysis. Parametric analysis is able to adapt bounds to the number of loop iterations, thereby more tightly bounding the actual number of required cycles for a task (Table II). Hence, it can save a significant number of cycles compared to numerical analysis (which must always assume the worst case – *i.e.* 1000 iterations in Figure 7). This effect becomes more pronounced as the number of actual iterations becomes much smaller than the static bound. In such situations, parametric timing analysis is able to provide significantly tighter bounds.

4. CREATION AND TIMING ANALYSIS OF FUNCTIONS THAT EVALUATE PARAMETRIC EXPRESSIONS

In the previous section, the methodology for deriving WCET bounds from parametric formulae was introduced. In this section, problems in embedding such formulae in application code are discussed. An iterative reevaluation of WCETs is provided as a solution.

The challenge of embedding evaluation functions for parametric formulae is as follows. When the code within a task is changed to include parametric WCET calculations, previous timing estimates and the caching behavior of the task might be affected. One may either inline the code of the formula or invoke a function that evaluates the symbolic formula. Since both approaches affect caching, another pass of cache analysis has to be performed on the modified code. We made an arbitrary design decision to pursue the latter approach. Using this modular approach, the cache analysis can reach a fixed point in fewer iterations as changes are constrained to functional boundaries rather than embedded within a function affecting the caching of any instructions below if the inlined code changes in size. The cost of calling an evaluation function is minimal compared to the benefit, and a subsequent call to the scheduler is required in any case to benefit from lower bounds.

Once a task has been enhanced with these parametric functions and their calls prior to loops, the timing analyzer must be reinvoked to analyze the newly enhanced code. This allows us to capture the WCET of generated functions and their invocations in the context of a task. Notice that any re-invocation of the timing analyzer potentially changes the parametric formulae and their corresponding functions such that we have to iterate through the timing analysis process. This is illustrated in Figure 8 where the process of generating formulae is presented. The iterative process converges to a fixed point when parametric

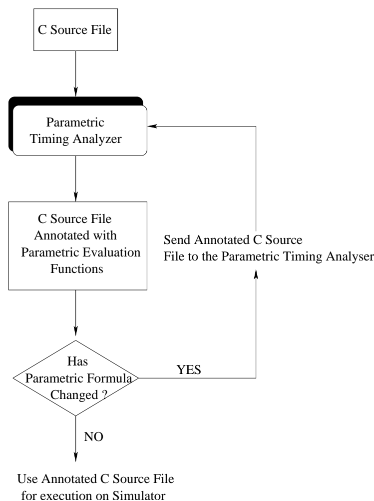
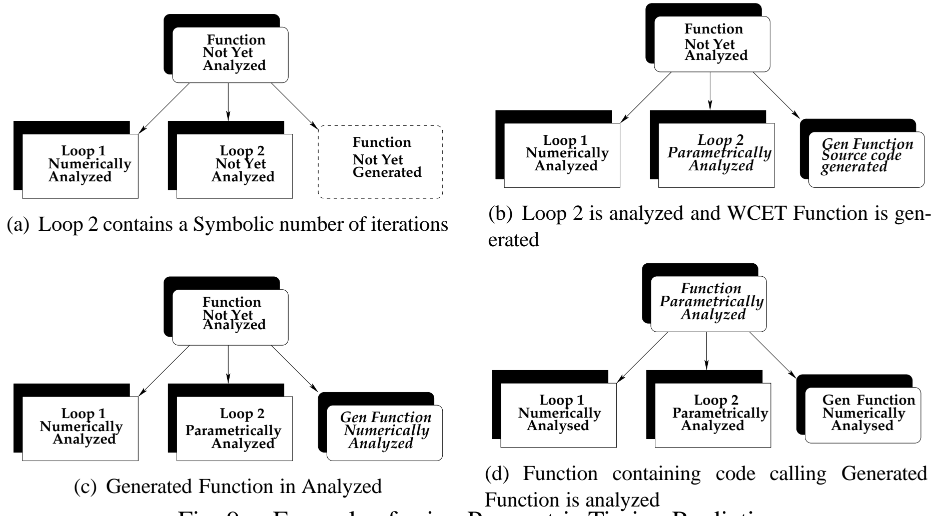


Fig. 8. Flow of Parametric Timing Analysis

formulae reach stable states. Typically, the parametric timing analysis and calculation of the parametric formulae take less than a second to complete. Since this is an offline process, it does not add to the overhead of the execution of the parametrized system.

An example is presented in Figure 9, where timing analysis is accomplished in stages, as parametric formulae are generated and evaluated later. In the example shown, a function



(a) Loop 2 contains a Symbolic number of iterations

(b) Loop 2 is analyzed and WCET Function is generated

(c) Generated Function in Analyzed

(d) Function containing code calling Generated Function is analyzed

Fig. 9. Example of using Parametric Timing Predictions

is generated by the timing analyzer to calculate the WCET for loop 2, whose number of iterations is only known at run-time.

The following sequence of operations takes place:

(1) A call to a function is inserted that returns the WCET for a specified loop or function based on a parameter indicating the number of loop iterations that is available at run time. The instructions that are associated with the call and the ones that use the return value after the call are generated during the initial compilation. For instance, in Figure 9(a) a function calls the yet-to-be generated function to obtain the WCET of loop 2, which contains a symbolic number of iterations.

(2) The timing analyzer generates the source code for the called function in a separate file when processing the specified loop or function whose time needs to be calculated at run time. For instance, Figure 9(c) shows that after loop 2 has been parametrically analyzed, the code for the calculating function has been generated. Note that the timing analysis tree representing the loops and functions in the program is processed in a bottom-up fashion. The code in the function invoking the generated function is not evaluated until after the generated function is produced. The static cache simulator can initially assume that a call to an unknown function invalidates the entire cache. Figure 3 shows an example of the source code for such a generated function.

(3) The generated function is compiled and placed at the end of the executable. The formula representing the symbolic WCET need not be simplified by the timing analyzer. Most optimizing compilers perform constant folding, strength reduction, and other optimizations that will automatically simplify the symbolic WCET produced by the timing analyzer. By placing the generated function after the rest of the program, instruction addresses of the program remain unaffected. While the caching behavior may have changed, loops are unaffected since timing tree is processed in a bottom-up order.

(4) The timing analyzer is invoked again to complete the analysis of the program, which now includes calculating the WCET of the generated function and the code invoking

this function. For instance, Figure 9(c) shows that the generated function has been numerically analyzed and Figure 9(d) shows that the original function has been parametrically analyzed, which now includes the numeric WCET required for executing the new function.

In short, this approach allows for timing analysis to proceed in stages. Parametric formulae are produced when needed and source code functions representing these formulae are produced, which are also subsequently compiled, inserted into the task code and analyzed. This process continues until a formula is obtained for the entire program or task.

5. USING PARAMETRIC EXPRESSIONS

In this section, potential benefits of parametric formulae and their evaluation functions are discussed. A more accurate knowledge of the remaining execution time provides a scheduler with information about additional slack in the schedule. This slack can be utilized in multiple ways:

—A dynamic admission scheduler can accept additional real-time tasks due to parametric bounds of the WCET of a task, which become tighter as execution progresses.

—Dynamic slack can also be used for dynamic voltage (and frequency) scaling (DVS) in order to reduce power.

In the remainder of the paper, the latter case will be detailed. Recall that parametric timing analysis involves the integration of symbolic WCET formulae as functions and their respective evaluation calls into a task's code. Apart from these inserted function calls, we also insert calls to transfer control to the DVS component of an optional dynamic scheduler *before* entering parametric loops, as shown in Figure 3. The parametric expressions are evaluated at run-time (using evaluation functions similar to the one in the figure) as knowledge of actual loops bounds becomes available. The newly calculated, tighter bound, on the execution time for the parametric loop is passed along to the scheduler. The scheduler is able to determine newly found dynamic slack by comparing worst-case execution cycles (WCECs) for that particular loop with the parametrically bounded execution time. The WCECs for each loop and the task as a whole are provided to the scheduler by the static timing analysis toolset. Static loop bounds for each loop are provided by hand. Automatic detection of bounds is subject to future work.

Dynamic slack originating from the evaluation of parametric expressions at run-time is discovered and can be exploited by the scheduler for admission scheduling or DVS (see above). Our work is unique in that we exploit early knowledge of parametric loop bounds, thus allowing us to tightly bound the overall execution of the *remainder* of the task. To this effect, we have developed an intra-task DVS algorithm to lower processor frequency and voltage. Another unique aspect of our approach is that every successive parametric loop that is encountered during the execution of the task potentially provides more slack and, hence, allows us to further scale down the processor frequency. This is in sharp contrast to past real-time schemes where DVS-regulated tasks are sped up as execution progresses, mainly due to approaching deadlines.

6. FRAMEWORK

An overview of our experimental framework is depicted in Figure 10. The instruction information fed to the timing analyzer is obtained from our P-compiler, which preprocesses

gcc-generated PISA assembly. The C source files are also fed simultaneously to both the static and the parametric timing analyzers. Safe (but, due to the parametric nature of loops, not necessarily tight) upper bounds for loops are provided as inputs to the static timing analyzer (STA). The worst-case execution times/cycles, for tasks as well as loops, provided by the STA are provided as input to a scheduler. The C source files are also provided to the PTA. The PTA produces source files annotated with parametric evaluation functions as well as calls to transfer control to the scheduler *before* entry into a parametric loop. These annotated source files form the task set for execution by the scheduler.

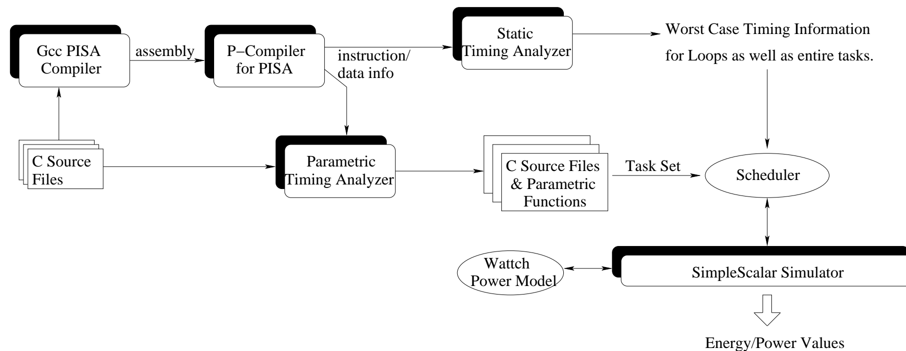


Fig. 10. Experimental Framework

To simplify the presentation, Figure 10 omits the loop that iterates over parametric functions till they reach a fixed point (as discussed in Figure 8). This would create a feedback between the PTA output and the C source files that provide the input to the toolset. For the sake of this discussion, we also combine the set of timing analysis tools as one component in Figure 10, *i.e.*, we omit the internal structure of a static cache simulator and the timing analyzer depicted in Figure 1.

We have implemented an EDF scheduler that creates an initial execution schedule based on the pessimistic WCET values provided by the STA. This scheduler is also capable of lowering the operating frequency (and, hence, the voltage) of the processor by way of its interaction with two DVS schemes: (a) an *inter-task* DVS algorithm, which scales down the frequency based on the execution of whole tasks (we use a *static* and a *dynamic* DVS algorithm) and (b) *ParaScale*, an intra-task DVS scheme that, on top of the scaled frequency from (a), which provides further opportunities to reduce the frequency based on dynamic slack gains due to PTA.

The static DVS scheme is similar to the static EDF policy by [Pillai and Shin 2001]. However, it differs in that the processor frequency and voltage are reduced to their respective minimum during idle periods. Two dynamic DVS schemes have been implemented. The first one, named “greedy DVS”, is a modification of the static DVS scheme and aggressively reduces the frequency below the statically determined value until the next scheduler invocation. The slack accrued from early completions of jobs is used to determine lower frequencies for execution.

The second dynamic DVS algorithm is the “lookahead” EDF-DVS policy by the same authors – it is a very aggressive dynamic DVS algorithm and lowers the frequency and voltage to very low levels. Throughout this paper, we shall use the name “ParaScale” to

refer to the intra-task DVS technique that uses the parametric loop information to accurately gauge the number of remaining cycles and lower the voltage/frequency. We shall use "ParaScale-G" and "ParaScale-L", to refer to the ParaScale implementations of the greedy and lookahead inter-task DVS algorithms, respectively. ParaScale always starts a task at the frequency value specified by the inter-task DVS algorithm. It then dynamically reduces the frequency and voltage according to slack gains from the knowledge on the re-calculated bounds on execution times for parametric loops. The effect of scaling is purely limited to intra-task scheduling, *i.e.*, the frequency can only be scaled down as much as the completion due to the non-parametric WCET allows. Hence, each call to the scheduler due to entering a parametric loop potentially results in slack gains and lower frequency/voltage levels.

We performed (numeric) timing analysis on the two schedulers in our system. The worst-case execution cycles for the schedulers (Table III) were then included in the utilization calculations. The WCEC for the inter-task DVS algorithm was used as a preemption overhead for all lower priority tasks. We assumed the worst-case behavior while dealing with preemptions, *i.e.*, the upper bound on the number of preemptions of a job j is given by the number of higher priority jobs released before job j's deadline.

The execution time for the intra-task DVS algorithm (ParaScale) was added *once* to the WCEC of each task in our system. The intra-task scheduler is called exactly once for each invocation of a task – prior to entry into the outermost parametric loop.

Scheduler Type	DVS Algorithm		
	no dvs	static dvs	lookahead dvs
Inter-task	6874	7751	8627
Intra-task	1625	2502	3378

Table III. WCECs for inter-task and intra-task schedulers for various DVS algorithms.

The simulation environment (used in a prior study [Anantaraman et al. 2003]) is a customized version of the SimpleScalar processor simulator that executes so-called PISA instructions (MIPS-like) [Burger et al. 1996]. PISA assembly, generated by gcc, also forms the input to the timing analyzers. The framework supports multitasking and the use of schedulers that operate with or without DVS policies. Our enhanced SimpleScalar is configured to model a static, in-order pipeline, with universal, unpipelined function units. We use a 64k instruction cache and *no data cache*. A static instruction cache simulator accurately models all accesses and produces categorizations, such as those illustrated in Table I. The data cache module has not been implemented yet, as our priority was to accurately gauge the benefits and energy savings of using parametric timing analysis. For the time being, we assume a constant memory access latency for each data reference and leave static data cache analysis for future work. Also, pipeline-related and cache-related preemption delays (CRPD) [Lee et al. 1996; Schneider 2000; Staschulat and Ernst 2004; Staschulat et al. 2005; Ramaprasad and Mueller 2006] are currently not modeled but, given accurate and safe CRPD bounds, could easily be integrated. The Wattch model [Brooks et al. 2000], along with the following enhancements, also forms part of the framework, in that it closely interacts with the simulator to assess the amount of power consumed. The original Wattch model provides power estimates assuming perfect clock gating for the units of the processor. An enhancement to the Wattch model provides more realistic results in that apart from perfect clock gating for the processor units, a certain amount of fixed leakage

is also consumed by units of the processor that are not in use. Closer examination of the leakage model of Wattch revealed that this estimation of static power may resemble but does not accurately model the leakage in practice. Static power is modeled by assuming that unused processor components leak approximately 10% of the dynamic power of the processor. This is inaccurate since static power is proportional to supply voltage while dynamic power is proportional to the *square* of the voltage. We discuss the effect of using the Wattch model in the following section. To reduce the inaccuracies of the Wattch model in determining the amount of leakage/static power consumed, we implemented a more accurate leakage model similar to prior work [Jejurikar et al. 2004]. The implementation is configurable so that we can not only study current trends for silicon technology (in terms of leakage), but we are also able to extrapolate on future trends (where leakage may dominate the total energy consumption of processors).

The minimum and maximum processor frequencies under DVS are 100MHz and 1GHz, respectively. Voltage/frequency pairs are loosely derived from the XScale architecture by extrapolating 37 pairs (five reported pairs between 1.8V/1GHz and 0.76V/150MHz) starting from 0.7V/100MHz in 0.03V/25MHz increments. Idle overhead is equivalent to execution at 100MHz, regardless of the scheduling scheme.

7. EXPERIMENTS AND RESULTS

We created several task sets using a mixture of floating-point and integer benchmarks from the C-Lab benchmark suite [C-Lab]. The actual tasks used are shown in Table IV. For

C Benchmark	Function	WCET	
		Cycles	Time [ms]
adpcm	Adaptive Differential Pulse Code Modulation	121,386,894	121.39
cnt	Sum and count of positive and negative numbers in an array	6,728,956	6.73
lms	An LMS adaptive signal enhancement	1,098,612	10.9
mm	Matrix Multiplication	67,198,069	67.2

Table IV. Task Sets of C-Lab Benchmarks and WCETs (at 1 GHz)

each task, the main control loop was parametrized. We had initially parametrized loops at all nesting levels, but we observed diminishing returns as the levels of nesting increased. In fact, the large number of calls to the parametric scheduler due to nesting had adverse effects on the power consumption relative to the base case. Hence, we limit parametric calls to outer loops only.

Table V depicts the period (equal to deadline) of each task. All task sets have the same

Utilization	Period = Deadline [ms]			
	adpcm	cnt	lms	mm
20%	1200	240	600	1200
50%	1200	75	60	600
80%	1200	50	40	240

Table V. Periods for Task Sets

hyperperiod of 1200 ms. All experiments executed for exactly one hyperperiod. This facilitates a direct comparison of energy values across all variations of factors mentioned in Table VI.

The parameters for the experiments are depicted in Table VI. We vary utilization, the

Parameter	Range of Values
Utilization	20%, 50%, 80%
Ratio WCET/PET	1x, 2x, 5x, 10x, 15x, 20x
Leakage Ratio	0.1, 1.0
DVS algorithms	Base Parametric Static DVS Greedy DVS ParaScale-G Lookahead ParaScale-L

Table VI. Parameters Varied in Experiments

ratio of worst-case to parametric execution times (PETs), and DVS support as follows:

Base: Executes tasks at maximum processor frequency and up to n, the actual number of loop iterations for parametric loops(not necessarily the maximum number of statically bounded iterations). The frequency is changed to the minimum available frequency during idle periods.

Parametric: Same as Base except that calls to the parametric scheduler are issued prior to parametric loops without taking any scheduling action. This assesses the overhead for scheduling of the parametric approach over the base case.

Static DVS: Lowers the execution frequency to the lowest valid frequency based on system utilization. For example, at 80% utilization, the frequency chosen would be 80% of the maximum frequency. Idle periods, due to early task completion, are handled at the minimum frequency.

Greedy DVS: This scheme is similar to static DVS in that it starts with the statically fixed frequency but then aggressively lowers the frequency for the current time period based on accrued slack from previous task invocations. Every time a job completes early, the slack gained is passed on to the job which follows immediately. Let job i be the job that completes early and generates slack and let job j be the job which follows (consumer). The greedy DVS algorithm calculates the frequency of execution, α', for j as follows:

$$\alpha' = \left[\frac{\alpha * C_j}{\alpha * C_j + slack_i}\right] \alpha \tag{3}$$

where α is the frequency determined by the static DVS scheme. Notice that *(a)* this slack is "lost" or rather reset to zero when the next scheduling decision takes place and *(b)* Equation 3 ensures that the new frequency scales down job j so that it attempts to completely utilize the slack from the previous job, but it does not stretch beyond the time originally budgeted for its execution based on the higher, statically determined, frequency. From (a) and (b) above, we see that the new DVS scheme will never miss a deadline if the original static DVS scheme never misses a deadline since greedy DVS accomplishes at least

the same amount of work as before, *i.e.*, it never utilizes processor time which lies beyond the original completion time of task j. The processor switches to the lowest possible frequency/voltage during idle time.

ParaScale-G: Combines the greedy and intra-task DVS schemes so that jobs start their execution at the lowest valid frequency based on system utilization. Before a parametric loop is entered, the frequency is scaled down further according to the difference between the WCET bound of the loop and the parametric bound of the loop calculated dynamically. ParaScale-G also exploits savings due to already completed execution relative to the WCET for frequency scaling. (These savings are small compared to the savings of parametric loops since parametric loops generally occur early in the code). It also utilizes job slack accrued from previous task invocations to further reduce the frequency. As in the case of the Static and Greedy DVS schemes, the processor switches to the lowest possible frequency/voltage during idle time.

Lookahead: Implements an enhanced version [Zhu and Mueller 2005] of Pillai's [Pillai and Shin 2001] *lookahead* EDF-DVS algorithm – a very aggressive dynamic DVS algorithm.

ParaScale-L: Combines the lookahead and intra-task DVS which utilizes parametric loop information. It is similar in operation to ParaScale-G. While ParaScale-G uses static values for initial frequencies, ParaScale-L uses frequencies calculated by the aggressive, dynamic EDF-DVS algorithm (lookahead).

Notice that all scheduling cases result in the *same amount of work* being executed during the hyperperiod (or any integer multiple thereof) due to the periodic nature of the real-time workload. Hence, to assess the benefits in terms of power awareness, we can measure the energy consumed over such a fixed period of time and compare this amount between scheduling modes.

The scheduler overhead for the greedy DVS scheme differs from those of the static DVS scheme by only a few cycles, as the only additional overhead is the calculation to determine α' (Equation 3). This calculation is performed only once per scheduler invocation because we only calculate the new frequency for the next scheduled task instance. Three types of energy measurements are carried out during the course of our experiments:

PCG: Energy used with *perfect* clock gating (PCG) – only processor units that are used during execution contribute to the energy measurements. This isolates the effect of the parametric approach on dynamic power.

PCGL: Energy consumed by leakage, *only*, based on prior methods [Jejurikar et al. 2004]. This attempts to capture the amount of energy exclusively used due to leakage.

PCGL-W: Energy used with perfect clock gating for the processor units including *leakage*. Leakage power is modeled by Wattch as 10% of dynamic power, which is not completely correct, as discussed before.

We also vary the ratio of worst-case to actual (parametric) execution times to study the effect of variations in execution times and make the experimental results more realistic. More often than not, the worst-case analysis of systems results in overestimations of WCET. ParaScale can take advantage of this to obtain additional energy savings.

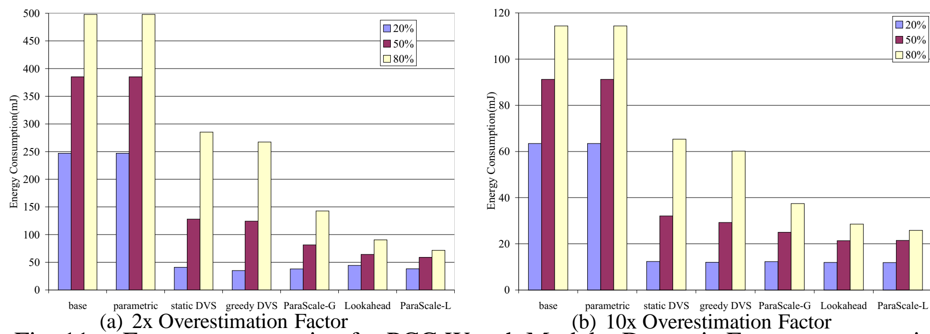
As part of the setup for the experiments we initialized the PCGL leakage model's operating parameters with the ratio of leakage to dynamic power for one particular experimental point. The ratio of dynamic and leakage energies for the WCET overestimation of 1x and

utilization of 50% was chosen for this purpose. This ratio was used to set up appropriate operating parameters (number of transistors, body bias voltage, *etc.*), after which the experiments were allowed to execute freely to completion. This gave us a unique opportunity to study the effects of leakage for *(a)* current processor technologies, where the ratio of leakage to dynamic may be 1:10 and *(b)* future trends where the leakage may increase significantly as the above ratio approaches 1:1. The "leakage ratios" mentioned in table VI refer to these two settings.

7.1 Overall Analysis

Figure 11 depicts the dynamic energy consumption for two sets of experiments – *(a)* Figure 11(a) shows the dynamic energy values for the case where the WCET overestimation is assumed to be twice that of the PET, and *(b)* Figure 11(b) shows the results for the instance where the WCET overestimation is assumed to be ten times that of the PET. Both graphs depict results for different utilization factors for each of the DVS schemes. From these graphs, we see that the energy consumption by the ParaScale implementations outperform their corresponding non-ParaScale implementations. Note that the greedy DVS scheme is able to achieve some savings relative to the static DVS scheme. These savings are fairly small, as the slack from the early completion of a job is passed on to the next scheduled job, if at all. ParaScale-G, on the other hand, is able to achieve *significant* savings over both the aggressive greedy algorithm and the static DVS algorithm. This shows that most of the savings of ParaScale-G is due to the early discovery of dynamic slack by the intra-task ParaScale algorithm.

ParaScale-L also shows *much* lower energy consumptions than the static DVS, greedy DVS, and the base case, always consuming the least amount of energy for all utilizations among the three DVS schemes. Note that higher relative savings are obtained for the higher utilization tasksets. This is true for all DVS schemes.



(a) 2x Overestimation Factor (b) 10x Overestimation Factor

Fig. 11. Energy consumption for PCG Wattch Model – Dynamic Energy consumption

Also, ParaScale-L outperforms the lookahead DVS algorithm, albeit by a small margin. The reason for this small difference is that lookahead is a very aggressive dynamic scheme, which tries to lower the frequency and voltage as much as possible and often executes at the lowest frequencies. ParaScale-L is able to outperform the lookahead algorithm due to the early discovery of future slack for parametric loops, which basic lookahead is unable to exploit fully.

One very interesting result is the relatively small difference between the ParaScale-G and the lookahead energy consumption results (for dynamic energy consumption). Thus,

ParaScale-G, an intra-task DVS scheme that enhances a *static* inter-task DVS scheme, results in energy savings that are close to those of the most aggressive *dynamic* DVS schemes, albeit at lower scheduling overhead of the static scheme.

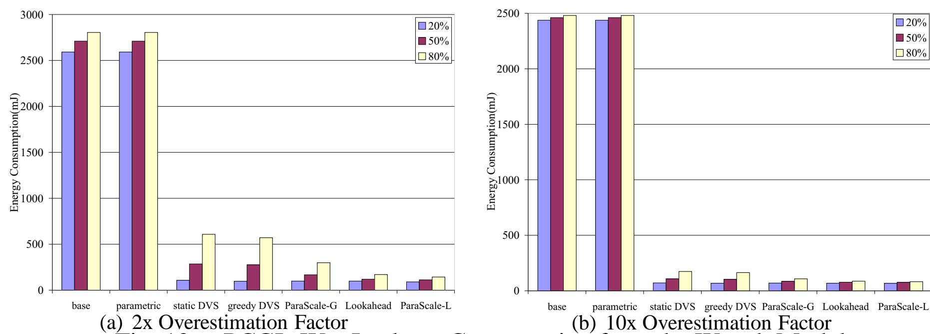
7.2 Leakage/Static Power

The results presented in Figure 11 are for energy values assuming perfect clock gating (PCG) within the processor, *i.e.*, they reflect the dynamic power consumption of the processor. These results isolate the *actual* gains due to the parametric approach. However, dynamic power is not the only source of power consumption on contemporary processors, which also have to account for an increasing amount of *leakage/static power* for inactive processor units.

In Figures 12 and 13, we present the energy consumed due to leakage. Figure 12 presents energy consumption with perfect clock gating and a constant leakage for function units that are not being utilized, as gathered by the Wattch power model. In reality, Wattch estimates the leakage to be 10% of the dynamic energy consumption at maximum frequency. This might not be entirely accurate. Even with this simplistic model, we see that the ParaScale implementations outperform all other DVS algorithms, as far as leakage is concerned. Notice that the absolute energy levels are very similar for 2x and 10x for the corresponding schemes. This is due to the dominating leakage in this case.

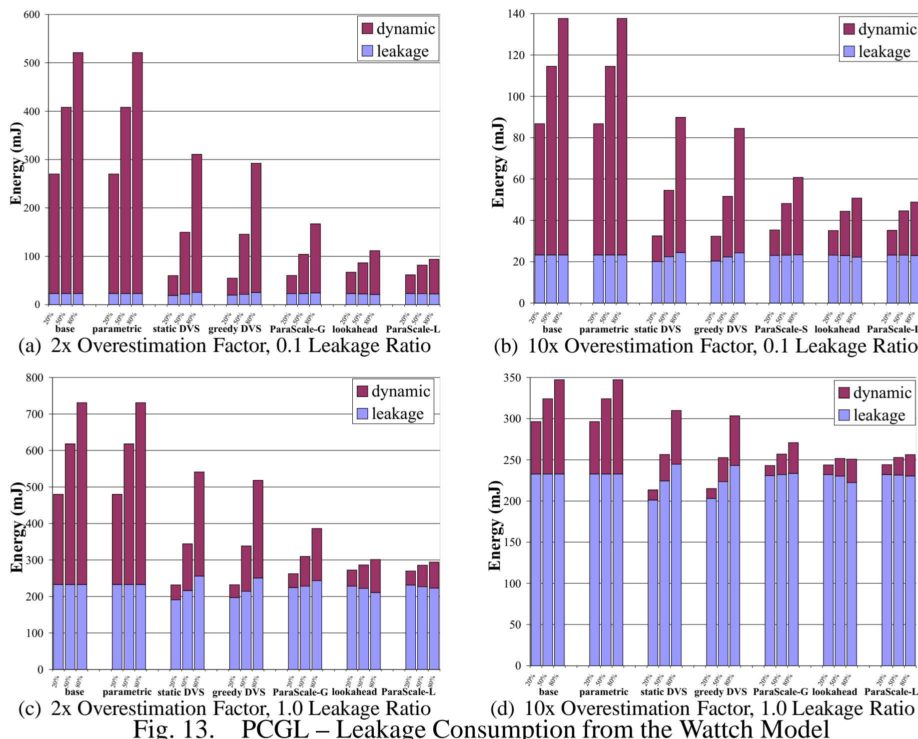
Figure 13 depicts leakage results for a more realistic and accurate leakage model similar to prior work [Jejurikar et al. 2004]. As mentioned earlier, we performed two sets of experiments with two ratios of leakage to dynamic energy consumptions – 0.1 and 1.0. While the former models current processor and silicon technologies, the latter extrapolates future trends for leakage. The top portions of the graphs in Figure 13 indicate the dynamic energy consumed while the lower portions indicate leakage. Figures 13(a) and 13(b) show the results for a leakage ratio of 0.1 for the 2x and 10x WCET overestimations respectively, and Figures 13(c) and 13(d) show similar results for a leakage ratio of 1.0.

From these graphs, we see that even when the leakage ratio is small, the leakage consumed might be a significant part of the total energy consumption of the processor. In fact, as Figure 13(b) shows, with a higher amount of slack in the system, the leakage could become dominant eventually accounting for more than half of the total energy consumption of the processor. Of course, Figures 13(c) and 13(d) show that even when the amount of slack in the system is low (2x WCET overestimation case), leakage might dominate energy consumption for future processors.



(a) 2x Overestimation Factor (b) 10x Overestimation Factor

Fig. 12. PCGL-W – Leakage Consumption from the Wattch Model



(a) 2x Overestimation Factor, 0.1 Leakage Ratio

(b) 10x Overestimation Factor, 0.1 Leakage Ratio

(c) 2x Overestimation Factor, 1.0 Leakage Ratio

(d) 10x Overestimation Factor, 1.0 Leakage Ratio

Fig. 13. PCGL – Leakage Consumption from the Wattch Model

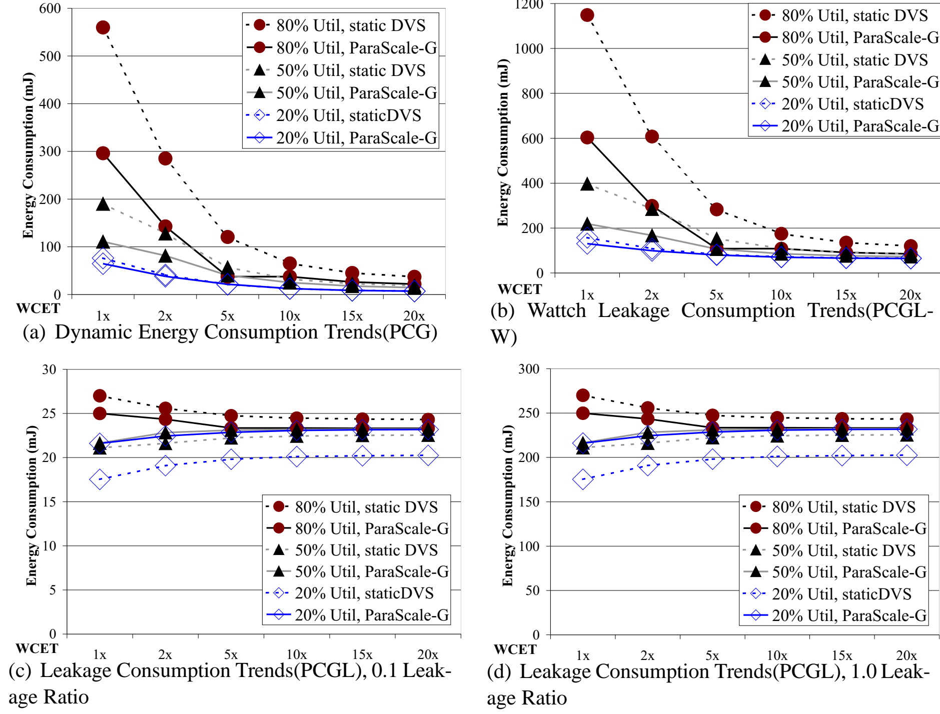
The ParaScale algorithms either outperform or are very close to their respective DVS algorithms (greedy DVS and lookahead) in all cases. The energy consumption of ParaScale-G often results in energy consumption similar to that of the dynamic lookahead DVS algorithm. This holds true for leakage as well as the total energy consumption (dynamic+leakage). Also, the combination of lookahead and the inter-task ParaScale (ParaScale-L) outperforms all other implementations.

The graphs in Figure 13 indicate identical static energy consumptions for all utilizations for the base and parametric experiments. The DVS algorithms, on the other hand, leak different amounts of static power for each of the utilizations. This effect is due to the fact that leakage depends on the actual voltage in the system. The static DVS algorithm consumes more leakage with increasing systems utilizations since it executes at higher, statically determined frequencies (and,hence,voltages) for higher utilizations. The greedy scheme performs *slightly* better as it is able to lower the frequency of execution due to slack passing between consecutive jobs. The lookahead and all ParaScale algorithms are able to aggressively lower their frequencies and voltage. Thus, they have a different leakage pattern compared to the constant values seen for the non-DVS cases or the increasing pattern for static DVS.

7.3 WCET/PET Ratio, Utilization Changes and Other Trends

We now consider the effects of changing the WCET overestimation factor and utilization on energy consumption. We shall use the ParaScale-G algorithm as a case study and compare it to static DVS and the base cases as depicted in Figures 11.

We observe slightly smaller relative energy savings for higher WCET factors (10x) com-



(a) Dynamic Energy Consumption Trends(PCG)

(b) Wattch Leakage Consumption Trends(PCGL-W)

(c) Leakage Consumption Trends(PCGL), 0.1 Leakage Ratio

(d) Leakage Consumption Trends(PCGL), 1.0 Leakage Ratio

Fig. 14. Energy Consumption Trends for increasing WCET Factors for ParaScale-G

pared to lower ones (2x). This is due to the fact that more slack is available in the system for the static algorithm to reduce frequency and voltage. Irrespective of the overestimation factor, ParaScale-L performs best for all utilizations, as discussed further in this section. The absolute energy level of 2x overestimation is about 3.5 times that of the 10x case without considering leakage for the highest utilization.

Furthermore, our technique performs better for higher utilizations, as seen for experiments with 80% utilization in Figure 11(a). As the ParaScale technique is able to take advantage of intra-task scheduling based on knowledge about past as well as future execution for a task, it is able to lower the frequency more aggressively than other DVS algorithms. This is more noticeable for higher utilization tasksets because less static slack is available to static algorithms for frequency scaling.

Figure 14 shows the trends in energy consumption across WCET/PET ratios ranging from 1x (no overestimation) to 20x. Energy values for both DVS algorithms — static DVS and ParaScale-G — are presented. In Figure 14(a), we see that energy consumption drops as the over-estimation factor is increased, since less work has to be done during the same time frame. We also see that the ParaScale-G algorithm is able to obtain more *dynamic* energy savings relative to the static DVS algorithm.

Similar trends exist in the results for PCGL-W (Figure 14(b)), except that the leakage, which permeates all experiments, results in lower relative savings compared to the PCG measurements. When contrasting Figure 14(a) to Figure 14(b), we observe that the overall energy consumption is higher in the latter. This is due to additional static power that is modeled by Wattch as 10% of dynamic power.

From the graphs for leakage (PCGL) shown in Figures 14(c) and 14(d), we see a more accurate modeling of leakage prevalent in the system. As the WCET overestimation factor is increased from 1x to 20x the leakage consumption trends appear similar, across the board, for both – ParaScale-G as well as static DVS . We observe that more and more the time is spent in idling(executing at the lowest frequency and operating voltage) and less in execution. The leakage energy increases slightly from 2x to 5x, but from there on remains nearly constant until 20x.

7.4 Comparison of ParaScale-G with Static DVS and Lookahead

We now present a comparison of ParaScale with greedy DVS and lookahead since the latter are two very effective DVS algorithms. Both algorithms have been implemented as stand-alone versions as well as hybrids integrated with ParaScale.

We already compared ParaScale-G to static DVS based on results provided in Figure 14. The energy consumption for ParaScale-G is significantly lower than that of static DVS across all experiments in Figure 14(a). This is because ParaScale-G can lower frequencies more aggressively over static DVS algorithms. Static DVS can only lower frequencies to statically determined values. We infer from Figure 14 that the relative savings drop in lower utilization systems and in systems with a high overestimation value. Due to the amount of static slack prevalent in such systems, the static DVS scheme is able to lower the frequency/voltage to a higher degree. For higher utilizations and for systems where the PETs match WCETs more closely, ParaScale-G is able to show the largest gain. This underlines one advantage of the ParaScale technique, *viz.* its ability to *predict dynamic slack* just before loops. This is particularly pronounced for higher utilization experiments resulting in lower energy consumption.

Consider the leakage results from Figure 14(b). We observe that the differences between the energy values for static DVS and ParaScale are much larger, especially for the lower utilization and higher WCET ratios. There exist two reasons for this result. (1) Static power depends on the voltage. When running at higher frequencies/voltages, as necessitated by higher utilizations, both static and dynamic power increases. (2) Static power is estimated to be 10% of the dynamic power by Wattch. Hence, higher utilizations with higher voltage and power values result in larger static power as well. This is compounded by the inaccurate modeling of leakage by the Wattch model. Dynamic power is proportional to the square of the supply voltage, whereas static power is directly proportional to the supply voltage. By assuming that static power accounts for 10% of power, Wattch makes the simplifying assumption that static power also scales quadratically with supply voltage.

Results from the more accurate leakage model are presented in Figures 14(c) and 14(d). We see that for the highest utilization (80%) ParaScale-G is able to lower the frequency and voltage enough so that the leakage energy dissipation is lower than that for static DVS. For the 50% and 20% utilizations, ParaScale-G shows a slightly worse performance. The leakage model that we used [Jejurikar et al. 2004] biases the per-cycle energy calculation with the inverse of the frequency (f^{-1}), which is the delay per cycle. Hence, aggressively lowering the frequency to the lowest possible levels may actually be counter-productive as far as leakage is concerned. The static DVS scheme lowers the frequency of execution to a lowest possible value of 200 MHz (for the 20% utilization experiments) while the ParaScale schedulers often hit the lowest frequency value (100 MHz). It is possible that the quadratic savings in energy due to a lower voltage are overcome by the increased delay per cycle at the lowest frequencies. Hence, if the number of execution cycles is large

enough, ParaScale experiments "leak" more energy than the static DVS scheme. Figure 13, though, shows that the *total* energy savings for the system is still lower for the ParaScale experiments compared to their equivalent non-ParaScale implementations, and ParaScale-L still consumes the least amount of energy.

Figure 15 depicts ParaScale-G, our inter-task DVS enhancement to the static DVS algorithm. It shows an energy signature that comes close to that of lookahead, one of the

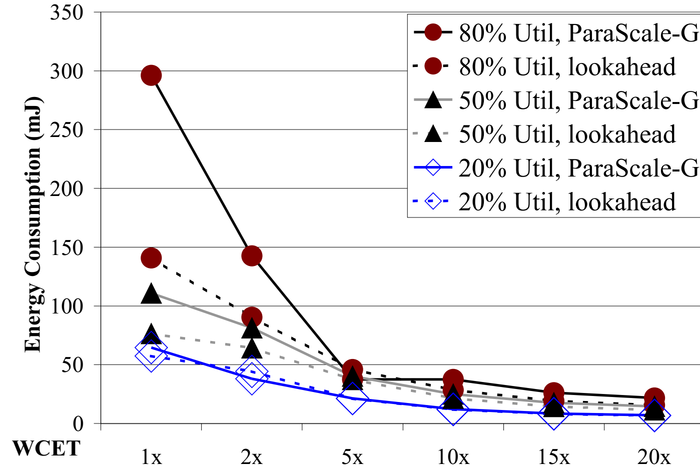


Fig. 15. Comparison of Dynamic Energy Consumption for ParaScale-G and Lookahead

best dynamic DVS algorithms. At times, ParaScale-G equals the performance of lookahead. This is particularly true for lower WCET factors where lookahead has less static and dynamic slack to play with. Here, ParaScale-G's performance is just as good, because it detects future slack on entry into parametric loops. This implies that we can achieve energy savings similar to those obtained by lookahead with a potentially lower algorithmic and implementation complexity. In fact, ParaScale-G is an $O(1)$ algorithm evaluating the parameters for only the *current* task whereas lookahead, an $O(n)$ algorithm traversing through all tasks in the system. This becomes more relevant as the number of tasks in the system is increased.

7.5 Overheads

The overheads imposed by the scheduler (especially the parametric scheduler, due to multiple calls made to it during task execution) and the frequency/voltage switching overheads are side-effects of the ParaScale technique. These scheduler overheads impose additional execution time on the system. The scheduler overheads were modeled using our timing analysis framework and are enumerated in Table III. When compared to the execution cycles for the tasks (Table IV) in the system, we see that the scheduler overheads are almost negligible when compared with task execution times. For example, the largest number of cycles used during a scheduler invocation is for the inter-task lookahead scheduler (8627 cycles). This value is less than 0.8% of the WCEC for the smallest task in the system, *viz.* LMS. Hence, the scheduler overheads have no significant impact on the execution of the tasks or the amount of energy savings.

7.5.1 *Frequency Switch Overheads.* To study the overheads imposed by the switching of frequencies and voltages, we imposed the overhead for a synchronous switch observed on an IBM PowerPC 405LP [Zhu and Mueller 2005]. The actual value used was $162\mu s$ for the overhead. We collected data on the number of frequency/voltage transitions for each experiment. The exact value of switching overhead varies depending on the actual difference between the voltages and whether it is being increased or decreased. We use this pessimistic, worst-case value to measure the worst possible switching overhead for the system. The highest overhead is incurred for the 20x overestimation case with utilization of 80% for ParaScale-G. The cumulative value for the overhead in this case was $42ms$. To put this in perspective, let us assume that the entire simulation had executed at the maximum frequency of 1 GHz. (thus completing in the shortest possible duration). The hyperperiod for each experiment was 1.2 seconds. All experiments were designed to execute for one hyperperiod. Since the tasksets execute at lower frequencies than the maximum, they will take longer to complete but still finish within their deadlines. Also, the frequency switch overhead is typically lower than $162\mu s$ (depending on the exact difference between the voltage/frequency levels). Hence, we can safely assume that the frequency switch overheads would be *much* less than the worst-case value of $42ms$. Typically, the overheads would be close to, or even less than, 1% of the total execution time of all tasks.

We also measured the energy consumption for the time period when the switching is taking place ($162\mu s$), for all three energy schemes – PCG, PCGL and PCGL-W. The respective values were 0.493 mJ, 0.007 mJ and 0.732 mJ, respectively, at 1 GHz. Considering the energy signature of the entire task set and the experiments, we can conclude that the energy overheads for frequency switching will be negligible.

8. RELATED WORK

Timing analysis has become an increasingly popular research topic. This can be attributed in part to the problem of increasing architectural complexity, which makes applications less predictable in terms of their timing behavior, but it may also be due to the abundance of embedded systems that we have recently seen. Often, application areas of embedded systems impose stringent timing constraints, and system developers are becoming aware of a need for verified bounds on execution times. While dynamic timing methods cannot provide safe bounds on the WCET, static timing analysis can [Wegener and Mueller 2001]. Nonetheless, dynamic bounds can complement static ones by providing a means to assess their tightness.

These developments are reflected in the research community where numerous methods for static timing analysis have been devised, ranging from unoptimized programs executing on simple CISC processors to optimized programs on pipelined RISC processors and even uncached architectures to instruction and data caches as well as branch prediction and locking caches [Park 1993; Puschner and Koza 1989; Harmon et al. 1992; Lim et al. 1994; Healy et al. 1995; Mueller 2000; White et al. 1999; Ferdinand and Wilhelm 1999; Lundqvist and Wall 1996; Li et al. 1996; Colin and Puaut 2001; Mitra and Roychoudhury 2002; Vera et al. 2003; Thesing et al. 2003].

In the past, path expressions were used to combine a source-oriented parametric approach of WCET analysis with timing annotations, verifying the latter with the former, particularly by Chapman *et al.* [Chapman et al. 1996]. Bernat and Burns proposed algebraic expressions to represent the WCET of programs [Bernat and Burns 2000]. Bernat *at*

el. used probabilistic approaches to express execution bounds down to the granularity of basic blocks that could be composed to form larger program segments [Bernat et al. 2002]. Yet, the combiner functions are not without problems, and timing of basic blocks requires architectural knowledge similar to static timing analysis tools.

Parametric timing analysis by Vivancos *et al.* [Vivancos et al. 2001] first introduced techniques to handle variable loop bounds as an extension to static timing analysis. That work focuses on the use of static analysis methods to derive parametric formulae to bound variable-length loops. Our work, in contrast, assesses the benefits of this work, particularly in the realm of power-awareness.

The effects of DVS on WCET have been studied in the FAST framework [Seth et al. 2003]. Here, parametrization was used to model the effect of memory latencies on pipeline stalls as processor frequency is varied. In our timing analyzer, we currently do not model these effects. This does not affect the correctness of our approach since WCET bounds are safe without such modeling, but they may not be tight, as shown in the FAST framework. Hence, the benefits of parametric DVS may even be better than what we report here.

The VISA framework suggested architectural enhancements to gauge progress of execution by sub-task partitioning and exploits intra-task slack with DVS techniques [Anantaraman et al. 2003; 2004]. Their technique did not exploit parametric loops. Our work, in contrast, takes advantage of dynamically discovered loop bounds and does not require any modifications at the micro-architecture level.

Lisper used polyhedral flow analysis to specify the iteration space of loop nests and express them as parametric integer programming problems to subsequently derive a parametric WCET formula suitable for timing analysis using IPET (Implicit Path Enumeration Technique) [Lisper 2003]. Recent work by Byhlin *et. al.* [Byhlin et al. 2005] underlines the importance of using parametric expressions to support WCET analysis in the presence of different modes of execution. They parametrize their WCET predictions for automotive software based on certain parameters, such as frame size. Their work focuses on studying the relationship between parameters unique to modes of execution and their effect on the WCET. Other work by Gheorghita *et al.* [Gheorghita et al. 2005] also promotes a parametric approach but at the level of basic blocks to distinguish different worst-case paths.. Our parametric expressions, predating any of this work, accurately bound the WCET values for *loops*. This extends the applicability of static analysis to a new class of programs. We take advantage of these accurate predictions at run-time for benefits such as power savings and admission of additional tasks. Tighter bounds on the WCET in the presence of DVS can also be achieved through a parametric model representing the latency in cycles to access main memory [Seth et al. 2003]. Due to DVS and constant memory access times, a lower processor frequency results in fewer cycles to access memory, which is reflected in WCET bounds in their FAST framework. This work is orthogonal to our method of PTA. In fact, our results could still be improved by employing FAST in the ParaScale context.

The most closely related work in terms of intra-task DVS is the idea of power management points (PMPs) [AbouGhazaleh et al. 2001; AbouGhazaleh et al. 2003; AbouGhazaleh et al. 2003]. In this work, path-dependent power management hints (PMHs) were used to aggregate knowledge about “saved” execution time compared to the worst-case execution that would have been imposed along different paths. This work differs in that it exploits knowledge about *past* execution while we discover loop bounds that let us provide tighter bounds on past and *future* execution within the same task. The work is also evaluated with

SimpleScalar, albeit with a more simplistic power model ($E = CV^2$) while we assess power at the micro-architecture level using enhancements of Wattch [Brooks et al. 2000] as well as a more accurate leakage model [Jejurikar et al. 2004]. Again, our results could potentially be improved by benefiting from knowledge about past execution, which may lead to additional power savings. This is subject to future work.

An intra-tasks DVS algorithm that "discovers" the amount of execution left in the system and appropriately modifies the frequency and voltage of the system is presented in [Shin et al. 2001]. Their work depends on inserting various instrumentation points at compile time into various paths in the code. Evaluation of these instrumentation points at runtime provides information about the paths taken during execution and the *possible* amount of execution time left along that path similar to PMPs. They insert instrumentation points in every basic block to determine the exact execution path, which would incur a significant overhead during runtime. This may also affect the caching and, hence, timing behavior of the task code. Our work differs significantly in that we only assess the amount of execution time remaining *once* (prior to entry into a parametric loop), thus incurring an overhead only once. We are, thus, able to accurately gauge the amount of execution remaining with a single overhead per loop and per task instance. We also estimate the new caching and timing behavior of the code after the call to the intra-task scheduler by invoking our timing analysis framework on the modified code until the parametric WCET formulae stabilize. Another technique presented in their paper is that of "L-type voltage scaling edges". They utilize the idea that loops are often executed for a smaller number of iterations than the worst-case scenario. During run-time, they discover the actual number of loop iterations at loop exit and then gauge the number of cycles saved. In contrast, parametric timing analysis determines loop savings *prior* to loop entry and exploits savings early, *e.g.*, using DVS, such as in ParaScale. This difference is a significant advantage for the parametric approach, particularly for tasks where a single loop nest accounts for most of the execution time.

9. CONCLUSION

In this paper, we (a) develop the novel technique of parametric timing analysis that obtains a formula to express WCET bounds, which is subsequently integrated into the code of tasks and (b) derive techniques to exploit parametric formulae *via* online scheduling and power-aware scheduling. We show how parametric formulae are integrated into the timing analysis process without sacrificing the tightness of WCET bounds. A fixed-point approach to embed parametric formulae into application code is derived, which bounds the WCET of not only the application code but also the embedded parametric functions and their calls once integrated into the application. Prior to entering parametric loops, the actual loop bounds are discovered and then used to provide WCET bounds for the remainder of execution of the tasks that are tighter than their static counterpart.

The benefit from parametric analysis is quantified in terms of power savings for sole intra-task DVS as well as ParaScale-G, our combined intra-task and greedy inter-task DVS. Processor frequency and voltage are scaled down as loop bounds of parametric loops are discovered. Power savings ranging between 66% to 80% compared to DVS-oblivious techniques are observed, depending on system utilization and the amount of overestimation for loop bounds. These energy savings are comparable to other DVS algorithms based on dynamic priority scheduling. Yet, our intra-task scheme lowers time complexity and can

be implemented as an extension to *static priority scheduling* or such as cyclic executives. Conventional timing analysis methods will be unable to achieve these benefits due to the lack of knowledge about remaining execution times of tasks in conventional static timing analysis. This illustrates the potential impact of PTA on the filed of timing analysis and real-time systems practitioners.

Overall, parametric timing analysis expands the class of applications for real-time systems to programs with dynamic loop bounds that are loop invariant while retaining tight WCET bounds and uncovering additional slack in the schedule.

REFERENCES

AbouGhazaleh, N., Childers, B., Mosse, D., Melhem, R., and Craven, M. 2003. Energy management for real-time embedded applications with compiler support. In *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*.

AbouGhazaleh, N., Mosse, D., Childers, B., and Melhem, R. 2001. Toward the placement of power management points in real time applications. In *Workshop on Compilers and Operating Systems for Low Power*.

AbouGhazaleh, N., Mosse, D., Childers, B., Melhem, R., and Craven, M. 2003. Collaborative operating system and compiler power management for real-time applications. In *IEEE Real-Time Embedded Technology and Applications Symposium*.

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley.

Al-Yaqoubi, N. 1997. Reducing timing analysis complexity by partitioning control flow. M.S. thesis, Florida State University.

Anantaraman, A., Seth, K., Patil, K., Rotenberg, E., and Mueller, F. 2003. Virtual simple architecture (VISA): Exceeding the complexity limit in safe real-time systems. In *International Symposium on Computer Architecture*. 250–261.

Anantaraman, A., Seth, K., Patil, K., Rotenberg, E., and Mueller, F. 2004. Enforcing safety of real-time schedules on contemporary processors using a virtual simple architecture (visa). In *IEEE Real-Time Systems Symposium*. 114–125.

Arnold, R., Mueller, F., Whalley, D. B., and Harmon, M. 1994. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*. 172–181.

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

Berg, C. 2006. Plru cache domino effects. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, F. Mueller, Ed. Number 06902 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. <http://drops.dagstuhl.de/opus/volltexte/2006/672> [date of citation: 2006-01-01].

Bernat, G. and Burns, A. 2000. An approach to symbolic worst-case execution time analysis. In *25th IFAC Workshop on Real-Time Programming*.

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

Brooks, D., Tiwari, V., and Martonosi, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. IEEE Computer Society and ACM SIGARCH, Vancouver, British Columbia, 83–94.

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

Byhlin, S., Ermedahl, A., Gustafsson, J., and per, B. L. 2005. Applying static wcet analysis to automotive communication software. In *ECRTS (Euromicro Conference on Real-Time Systems)*.

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

Chapman, R., Burns, A., and Wellings, A. 1996. Combining static worst-case timing analysis and program proof. *Real-Time Systems 11,* 2, 145–171.

CHEN, K., MALIK, S., AND AUGUST, D. I. 2001. Retargetable static timing analysis for embedded software. In *Proceedings of the International Symposium on System Synthesis (ISSS)*.

COLIN, A. AND PUAUT, I. 2001. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems 18,* 2/3, 249–174.

ENGBLOM, J. 2002. Processor pipelines and static worst-case execution time analysis. Ph.D. thesis, Dept. of Information Technology, Uppsala University.

ENGBLOM, J., ERMEDAHL, A., SJDIN, M., GUSTAFSSON, J., , AND HANSSON, H. 2001. Execution-time analysis for embedded real-time systems. In *STTT (Software Tools for Technology Transfer) special issue on ASTEC*.

FERDINAND, C. AND WILHELM, R. 1999. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems 17,* 2/3 (Nov.), 131–181.

GHEORGHITA, V. S., STUIJK, S., BASTEN, T., AND CORPORAAL, H. 2005. Automatic scenario detection for improved wcet estimation. In *Design Automation Conference*.

GOVIL, K., CHAN, E., AND WASSERMAN, H. 1995. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *1st Int'l Conference on Mobile Computing and Networking*.

GRUIAN, F. 2001. 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*.

GRUNWALD, D., LEVIS, P., III, C. M., NEUFELD, M., AND FARKAS, K. 2000. Policies for dynamic clock scheduling. In *Symp. on Operating Systems Design and Implementation*.

HARMON, M., BAKER, T. P., AND WHALLEY, D. B. 1992. A retargetable technique for predicting execution time. In *IEEE Real-Time Systems Symposium*. 68–77.

HEALY, C., SJODIN, M., RUSTAGI, V., WHALLEY, D., AND VAN ENGELEN, R. 2000. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems 18,* 2/3 (May), 121–148.

HEALY, C. A., ARNOLD, R. D., MUELLER, F., WHALLEY, D., AND HARMON, M. G. 1999. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers 48,* 1 (Jan.), 53–70.

HEALY, C. A., SJÖDIN, M. ., AND WHALLEY, D. B. 1998. Bounding loop iterations for timing analysis. In *IEEE Real-Time Embedded Technology and Applications Symposium*. 12–21.

HEALY, C. A., WHALLEY, D. B., AND HARMON, M. G. 1995. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*. 288–297.

HERGENHAN, A. AND ROSENSTIEL, W. 2000. Static timing analysis of embedded software on advanced processor architectures. In *DATE*. 552–559.

JEJURIKAR, R. AND GUPTA, R. 2005. Dynamic slack reclemation with procrastination scheduling in real-time embedded systems. In *Design Automation Conference*.

JEJURIKAR, R., PEREIRA, C., AND GUPTA, R. 2004. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Design Automation Conference*.

KANG, D., CRAGO, S., AND SUH, J. 2002. A fast resource synthesis technique for energy-efficient real-time systems. In *IEEE Real-Time Systems Symposium*.

LEE, C., HAHN, J., SEO, Y., MIN, S., HA, R., HONG, S., PARK, C., LEE, M., AND KIM, C. 1996. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *IEEE Real-Time Systems Symposium*.

LEE, C.-H. AND SHIN, K. G. 2004. On-line dynamic voltage scaling for hard real-time systems using the edf algorithm. In *IEEE Real-Time Embedded Technology and Applications Symposium*.

LEE, Y.-H. AND KRISHNA, C. M. 2003. Voltage-clock scaling for low energy consumption in fixed-priority real-time systems. *Real-Time Syst. 24,* 3, 303–317.

LI, Y.-T. S., MALIK, S., AND WOLFE, A. 1996. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*. 254–263.

LIM, S.-S., BAE, Y. H., JANG, G. T., RHEE, B.-D., MIN, S. L., PARK, C. Y., SHIN, H., AND KIM, C. S. 1994. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*. 97–108.

LISPER, B. 2003. Fully automatic, parametric worst-case execution time analysis. In *WCET*. 99–102.

LIU, C. AND LAYLAND, J. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Association for Computing Machinery 20,* 1 (Jan.), 46–61.

LIU, Y. AND MOK, A. K. 2003. An integrated approach for applying dynamic voltage scaling to hard real-time systems. In *Proceedings of the ninth IEEE Real-Time and Embedded Technology and Applications Symposium*.

LUNDQVIST, K. AND WALL, G. 1996. Using object oriented methods in Ada 95 to implement linda. In *Ada Europe*.

MALIK, S., MARTONOSI, M., AND LI, Y.-T. S. 1997. Static timing analysis of embedded software. In *Proceedings of the 34th Conference on Design Automation (DAC-97)*. ACM Press, NY, 147–152.

MITRA, T. AND ROYCHOUDHURY, A. 2002. A framework to model branch prediction for wcet analysis. In *2nd Workshop on Worst Case Execution Time Analysis (WCET)*.

MOHAN, S., MUELLER, F., HAWKINS, W., ROOT, M., HEALY, C., AND WHALLEY, D. 2005. Parascale: Expoliting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *IEEE Real-Time Systems Symposium*. 233–242.

MOSSE, D., AYDIN, H., CHILDERS, B., AND MELHEM, R. 2000. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low Power*.

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

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

PERING, T., BURD, T., AND BRODERSEN, R. 1995. The simulation of dynamic voltage scaling algorithms. In *Symp. on Low Power Electronics*.

PILLAI, P. AND SHIN, K. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Symposium on Operating Systems Principles*.

PUSCHNER, P. AND KOZA, C. 1989. Calculating the maximum execution time of real-time programs. *Real-Time Systems 1,* 2 (Sept.), 159–176.

RAMAPRASAD, H. AND MUELLER, F. 2006. Bounding preemption delay within data cache reference patterns for real-time tasks. In *IEEE Real-Time Embedded Technology and Applications Symposium*. 71–80.

SAEWONG, S. AND RAJKUMAR, R. 2003. Practical voltage-scaling for fixed-priority rt-systems. In *Proceedings of the ninth IEEE Real-Time and Embedded Technology and Applications Symposium*.

SCHNEIDER, J. 2000. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In *IEEE Real-Time Systems Symposium*. 195–204.

SETH, K., ANANTARAMAN, A., MUELLER, F., AND ROTENBERG, E. 2003. Fast: Frequency-aware static timing analysis. In *IEEE Real-Time Systems Symposium*. 40–51.

SHIN, D., KIM, J., AND LEE, S. 2001. Intra-task voltage scheduling for low-energy hard real-time applications. In *IEEE Design and Test of Computers*.

SHIN, Y., CHOI, K., AND SAKURAI, T. 2000. Power optimization of real-time embedded systems on variable speed processors. In *Int'l Conf. on Computer-Aided Design*.

STASCHULAT, J. AND ERNST, R. 2004. Multiple process execution in cache related preemption delay analysis. In *International Conference on Embedded Sofware*.

STASCHULAT, J., SCHLIECKER, S., AND ERNST, R. 2005. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Euromicro Conference on Real-Time Systems*.

THESING, S., SOUYRIS, J., HECKMANN, R., ANDM. LANGENBACH, F. R., WILHELM, R., AND FERDINAND, C. 2003. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics. In *Proceedings of the International Performance and Dependability Symposium (IPDS)*.

UNGER, S. AND MUELLER, F. 2002. Handling irreducible loops: Optimized node splitting vs. dj-graphs. *ACM Transactions on Programming Languages and Systems 24,* 4 (July), 299–333.

VERA, X., LISPER, B., AND XUE, J. 2003. Data caches in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*.

VIVANCOS, E., HEALY, C., MUELLER, F., AND WHALLEY, D. 2001. Parametric timing analysis. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Embedded Systems*. ACM SIGPLAN Notices, vol. 36. 88–93.

WEGENER, J. AND MUELLER, F. 2001. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems 21,* 3 (Nov.), 241–268.

WEISER, M., WELCH, B., DEMERS, A., AND SHENKER, S. 1994. Scheduling for reduced cpu energy. In *1st Symp. on Operating Systems Design and Implementation*.

WHITE, R., MUELLER, F., HEALY, C., WHALLEY, D., AND HARMON, M. 1997. Timing analysis for data caches and set-associative caches. In *IEEE Real-Time Embedded Technology and Applications Symposium*. 192–202.

WHITE, R. T., MUELLER, F., HEALY, C., WHALLEY, D., AND HARMON, M. G. 1999. Timing analysis for data and wrap-around fill caches. *Real-Time Systems 17,* 2/3 (Nov.), 209–233.

ZHANG, F. AND CHANSON, S. T. 2002. Processor voltage scheduling for real-time tasks with non-preemptable sections. In *IEEE Real-Time Systems Symposium*.

ZHONG, X. AND XU, C.-Z. 2005. Energy-aware modeling and scheduling of real-time tasks for dynamic voltage scaling. In *IEEE Real-Time Systems Symposium*.

ZHU, Y. AND MUELLER, F. 2004. Feedback edf scheduling exploiting dynamic voltage scaling. In *IEEE Real-Time Embedded Technology and Applications Symposium*. 84–93.

ZHU, Y. AND MUELLER, F. 2005. Feedback edf scheduling exploiting hardware-assisted asynchronous dynamic voltage scaling. In *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*. 203–212.

Received November 2005; revised April 2006; accepted September 2007