ParaScale: Exploiting Parametric Timing Analysis for Real-Time Schedulers and Dynamic Voltage Scaling *

Sibin Mohan[1], Frank Mueller[1], William Hawkins[2], Michael Root[3], Christopher Healy[3] and David Whalley[4]
[1] Dept. of Computer Science, Center for Embedded Systems Research,
North Carolina State University, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu
[2] Dept. of Computer Science, University of Illinios at Urbana-Champaign, Urbana, IL 61801, whawkin2@cs.uiuc.edu
[3] Dept. of Computer Science, Furman University, Greenville, SC 29613, chris.healy@furman.edu
[4] Dept. of Computer Science, Florida State University, Tallahassee, FL 32306, whalley@cs.fsu.edu

Abstract

Static timing analysis safely bounds worst-case execution times to determine if tasks can meet their deadlines in hard real-time systems. However, conventional timing analysis requires that the upper bound of loops be known statically, which limits its applicability. Parametric timing analysis methods remove this constraint by providing the WCET as a formula parameterized on loop bounds.

This paper contributes a novel technique to allow parametric timing analysis to interact with dynamic real-time schedulers. By dynamically detecting actual loop bounds, a lower WCET bound can be calculated, on-the-fly, for the remaining execution of a task. We analyze the benefits from parametric analysis in terms of dynamically discovered slack in a schedule. We then assess the potential for dynamic power conservation by exploiting parametric loop bounds for ParaScale, our intra-task dynamic voltage scaling (DVS) approach. Our results demonstrate that the parametric approach to timing analysis provides 66%-80% additional savings in power consumption. We further show that using this approach combined with online intra-task DVS to exploit parametric execution times results in much lower power consumption. Hence, even in the absence of dynamic scheduling, significant savings in power can be obtained, e.g.*, in the case of cyclic executives.*

1. Introduction

Embedded systems are increasingly deployed in environments where safety is of the utmost concern, ranging from avionics to power plants to the automotive industry. Validation of software in such systems is of increasing importance due to this trend. Program validation traditionally concerns the correctness of the input/output relation. In addition to I/O correctness, many embedded systems also impose timing constraints, which, if violated, may not only render a system nonfunctional, but is also dangerous to the environment. These systems are commonly referred to as real-time systems. They impose deadlines on computational tasks to ensure that results are supplied on time. Often, an approximate result supplied on time is preferably to a more precise result that becomes available late, *i.e.*, after the deadline. In order to verify that real-time systems will meet their deadlines, designers require that the worst-case execution time (WCET) of each task in a real-time system be known. The process of automatically and statically determining the WCET of a program or task is called timing analysis. Scheduling decisions are based on each task's WCET and the total time in the schedule.

Timing analysis provides bounds on the WCET, and the closer these bounds are to the true worst-case, the higher the value of the analysis. But timing analysis is by no means an easy task. Any bound on execution time first of all requires constraints to be imposed on the timed code (tasks). The most striking restriction is the requirement to statically bound the number of iterations within loops. Loop bounds are required to address the halting problem, *i.e.*, without loop bounds, the WCET could not be bounded. Upper bounds on loops can be supplied by the programmer or, when possible, inferred by program analysis. In practice, statically fixed loop bounds not only present an inconvenience to programmers who may have to specify them, they also restrict the programs that worst-case timing analysis can bound.

Parametric timing analysis (PTA) [42] lifts this requirement. Instead of statically known loop bounds, variable-length loops are permitted, *e.g.*, loops can be bounded by n iterations. The only restriction imposed is that the loop bound n be known prior to entering the loop during execution. Such a relaxation considerably widens the scope of analyzable programs and makes timing analysis more attractive for real-world embedded applications.

*This work was supported in part by NSF grants CCR-0208581, CCR-0310860, CCR-0312695, EIA-0072043, CCR-0208892, CCR-0312493 and CCR-0312531.

Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)
0-7695-2490-7/05 $20.00 © 2005 IEEE

Past work on PTA focused on the challenges of timing analysis to derive parametric expressions that bound the WCET of parametric loops as polynomial functions. The values affecting the execution time, such as loop bound n, are the parameters to such a function.

This paper focuses on assessing the benefits of PTA for online scheduling and, most significantly, for dynamic voltage scaling. Our work contributes a novel technique to let PTA interact with a dynamic scheduler. When discovering actual loop bounds during execution prior to entering a loop, a lower WCET bound can be calculated on-the-fly. This tighter bound on the remaining execution overhead of a task may then allow scheduling decisions to be triggered synchronously with the execution of the task. We analyze these benefits of PTA resulting from dynamically discovered slack.

Slack may not only be utilized for execution of additional tasks in admission scheduling, it can also be exploited for power management. Recently, numerous approaches to dynamic voltage scaling (DVS) methods have been studied, both for general-purpose systems [16, 18, 34, 44] and as well as real-time systems [17, 39, 35, 6, 14, 6, 22, 46, 37, 23, 27]. The core voltage of contemporary embedded processors can be reduced when also lowering their frequency. At a lower execution rate, power is significantly reduced since power is proportional to the frequency and to the square of the voltage: $P \propto V^2 \times f$.

Past real-time scheduling algorithms have exploited static and dynamic slack in inter-task DVS approaches [17, 39, 35, 6, 22, 46, 37, 23, 27] as well as intra-task saving schemes [31, 14, 6, 2]. Dynamic slack discovery is typically based on early task completion or assessing the progress of execution based on past executions.

Our work takes a novel approach. Instead of depending on savings from past execution, slack can be *safely predicted for future execution*. To this extent, we exploit early knowledge of parametric loop bounds, which allows us to more tightly bound the remainder of execution of a task. We then assess the potential for dynamic power conservation via *ParaScale*, our intra-task DVS algorithm. ParaScale allows tasks to be *slowed down* when more slack becomes available, in contrast to past real-time DVS schemes where tasks were actually sped up in later stages as they approach their deadline [17].

To evaluate ParaScale, we perform PTA on MIPS-like assembly generated by gcc. We then assess the benefits of exploiting PTA in the context of DVS by simulating execution within a customized SimpleScalar framework [10] that supports multi-tasking, allows the specification of a custom scheduler with or without DVS policies, and supports an assessment of consumed power through enhanced Wattch power models [9]. Our results demonstrate that the parametric approach to timing analysis, such as in ParaScale, provides significant savings, not only in terms of generating dynamic slack but also in its potential for power savings. We further show that using this approach combined with online intra-task DVS to exploit parametric execution times results in much lower power consumption by itself, *i.e.*, without any scheduler-assisted DVS savings. Hence, even in the absence of dynamic scheduling, significant savings in power can be obtained, *e.g.*, in the case of cyclic executives.

The paper is structured as follows. Sections 2 and 3 provide information on static as well as parametric timing analysis. Section 4 discusses the context in which parametric timing results are used. Section 5 introduces the simulation framework. Section 6 elaborates on the experiments and results that we have obtained. Section 7 discusses related work, and Section 8 summarizes the work.

2. Timing Analysis

Knowledge of worst-case execution times (WCETs) is necessary for most hard real-time systems. The WCET must be known and safely bounded, so that the feasibility of scheduling task sets in the system may be determined, given a scheduling policy, such as rate-monotone or earliest-deadline-first scheduling [26]. 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 [43]. 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 the following, we constrain ourselves to a toolset developed in our previous work [20, 32, 45, 30]. Static timing analysis models the traversal of all possible execution paths in the code. Execution timing is determined independent of program traces or values of 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 informa-

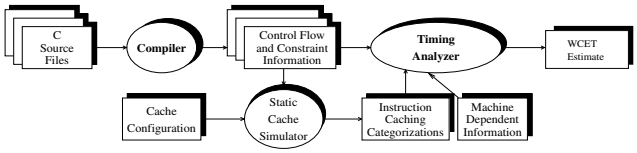


Figure 1. Static Timing Analysis Framework

Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)
0-7695-2490-7/05 $20.00 © 2005 IEEE

tion, 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 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 construct a control-flow graph of the program and caching categorizations for each instruction. 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. Each loop level containing the instruction and data references is analyzed to obtain separate categorizations. These categorizations for instruction references are described in Table 1.

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

Table 1. Instruction Categories for WCET

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), branch prediction 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.

WCET bounds for each path, then each loop, for each function and finally for the entire task is then derived by the timing analyzer by the construction of a timing tree, which is processed in a bottom up manner. WCET's 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) [42], in contrast, makes it possible to support timing predictions when the number of iterations for a loop is not known until run-time.

```
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);
}
```

Figure 2. Use of Parametric Timing Analysis

Consider the example in Figure 2. 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.

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.*, 1000 in this case.

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 2], 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.

Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)
0-7695-2490-7/05 $20.00 © 2005 IEEE

```
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);
```

Figure 3. Parametric Loop Analysis Algorithm

The algorithm in Figure 3 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, before the final worst-case path time is obtained, is then saved. It can subsequently be used to detect the number of cycles in a loop as follows:

$$WCET_{loop} = WCET_{path} * n + base\_cycles \quad (1)$$

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 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.

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, *via parametric analysis*, if the depth can be inferred dynamically prior to the first function call. 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.

Based on these symbolic formulae for loops, a scheduler can dynamically adjust the schedule based on the parametric estimate of the WCET of a task. Additional tasks may be introduced, and the voltage and frequency can also be reduced to save power.

If the code within a task is changed to include symbolic WCET formulae as well as calls to calculate these formulae, then previous timing estimates as well as caching behavior of the task might be changed. Hence, rather than inserting a formula directly into the source code at the point of usage, a function is invoked that evaluates the symbolic formula.

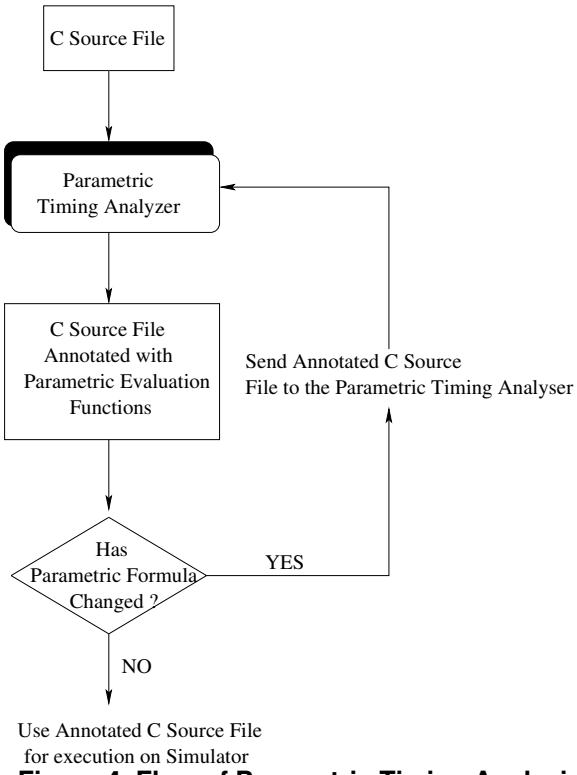


Figure 4. Flow of Parametric Timing Analysis

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 4 where the process of generating formulae is iterated. The iterative process converges to a fixed point when parametric 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 parameterized system.

4. Using Parametric Expressions

In the previous section, we illustrated the process of embedding parametric expressions and associated evaluations functions into the code of tasks. 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 2. 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

Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)
0-7695-2490-7/05 $20.00 © 2005 IEEE

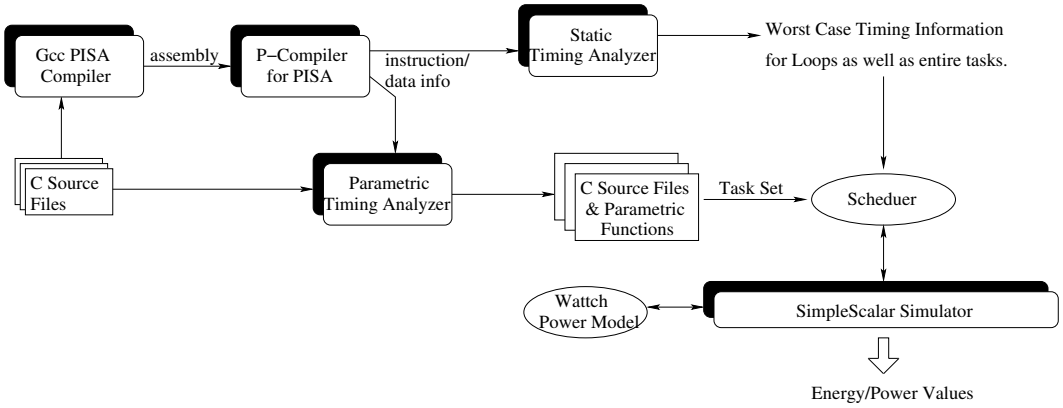


Figure 5. Experimental Framework

is able to calculate savings in execution time by comparing WCECs for that particular loop with the newly available 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.

Savings gained from the evaluation of parametric expressions at run-time can be exploited by the scheduler. The slack gained from the reduced execution can be used to admit additional tasks. The scheduler may also reduce the operating frequency and voltage (DVS) of the processor to save power. In our work, we use the knowledge of reduced execution times to reduce the voltage and operating frequency of the processor.

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 slow down the processor. 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 slow down the processor. 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.

5. Framework

An overview of our experimental framework is depicted in Figure 5. In addition to the typical instruction and data information fed to any timing analyzer, in this case obtained from the gcc-generated PISA assembly by a P-compiler, C source files are 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* the entry of any parametric loop. These annotated source files form the task set for execution by the scheduler. To simplify the presentation, Figure 5 omits the loop that iterates over parametric functions till they reach a fixed point (as discussed in Figure 4). 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 5, *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) a *static* DVS scheme that uniformly scales down frequency based on static slack and (b) *ParaScale*, an intra-task DVS scheme that, on top of the uniformly scaled frequency from (a), 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 [35]. However, it differs in that the processor frequency and voltage are reduced to their respective minimum during idle periods. ParaScale, our parametric DVS algorithm, starts a task at the static frequency value. It then dynamically reduces the frequency and voltage according to slack gains from the knowledge on the recalculated 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.

The simulation environment (used in a prior study [4]) is a customized version of the SimpleScalar processor simu-

Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)
0-7695-2490-7/05 $20.00 © 2005 IEEE

lator that executes so-called PISA instructions (MIPS-like) [10]. 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 1. The data cache module has not been modeled, 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. The Wattch model [9], along with certain 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 power is also consumed by units of the processor that are not in use.

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.

6. Experiments and Results

We created several task sets using a mixture of floating-point and integer benchmarks from the C-Lab benchmark suite [11]. The actual tasks used are shown in Table 2. For each of the tasks, the main control loop was parameterized. We had initially parameterized 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 3 depicts the period (equal to deadline) of each task. All task sets have the same 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 4.

The parameters for the experiments are depicted in Table 4. We vary utilization, the ratio of worst-case to parametric execution times (PETs), and DVS support as follows:

Base executes tasks at maximum processor frequency and

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

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

Table 3. Periods for Task Sets

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

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.

Parametric is the 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 in scheduling of the parametric approach over the base case.

ParaScale combines static and intra-task DVS so that tasks

Parameter	Range of Values
Utilization	20%, 50%, 80%
Ratio WCET/PET	1x, 2x, 5x, 10x, 15x, 20x
DVS algorithms	Base Static DVS Parametric ParaScale

Table 4. Parameters Varied in Experiments

Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)
0-7695-2490-7/05 $20.00 © 2005 IEEE

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 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.)

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. Two 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 used with perfect clock gating for the processor units and a *leakage* (PCGL) value for the remainder of the processor. This illustrates the effect of static power, which is becoming more and more important for smaller fabrication (die) sizes.

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.

Overall Trends

Figure 6 depicts the total energy consumption for each experiment. While Figure 6(a) depicts the energy consumption for the case where the WCET overestimation is assumed to be twice that of PET, Figure 6(b) depicts the energy consumption for an overestimation factor of ten. Both graphs break down the data into different utilization factors for the implementations. In both graphs, we see that the energy consumption by the ParaScale implementation is *much* lower than for the base or static DVS cases. In fact, ParaScale *always* consumes the least amount of energy for any given utilization.

From both graphs, we see significant savings while using our ParaScale model over the Base case, ranging from 66% for Figure 6(b) to 80% for 6(a). The savings over static are 16-60% for 2x under 20% and 80% utilization, respectively.

Leakage Power

The results presented in Figure 6 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 usually have some *leakage power* caused by inactive processor units. In Figure 7, we present energy consumption with perfect clock gating and a constant leakage power (PCGL) for function units that are not being used. We still see that results for experiments that use the parametric formulae outperform those that do not. Most noticeable are the results for ParaScale (at the right end), which uses a combination of static DVS and intra-task parametric DVS. We observe the least amount of energy consumption for these cases under all utilization scenarios. ParaScale, our combined static and parametric DVS, in contrast, outperforms all others by significant margins.

WCET/PET Ratio and Utilization

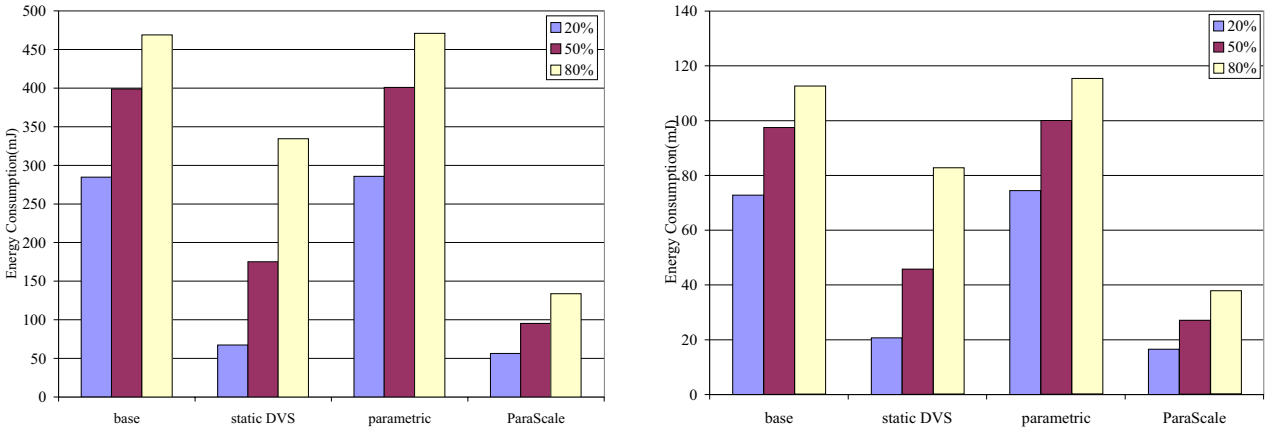
We observe slightly smaller energy savings for higher WCET factors (10x) when compared to lower ones (2x). This is due to the fact that more slack is available in the system for the static algorithm to reduce frequency and voltage. Irrespective of the overestimation factor, ParaScale performs the best for all utilizations. Another useful result is that our technique performs better for higher utilizations, as seen for the results for the experiments with 80% utilization in Figures 6 and 7. 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.

Parametric Scheduler Overhead

In Figures 6 and 7, we observe a striking similarity between results for base and parametric cases, but parametric results are *slightly* worse. In the parametric case, there is additional overhead in calling the parametric scheduler before entering into parametric loops while that both sets of experiments are identical in every other regard. Hence, the difference illustrates the low overhead of such additional scheduler calls. As mentioned before, we also experimented with nested parametric loops, which showed significantly higher energy values of the parametric case when compared to the base counterparts. Hence, nested parametric loops were considered an impractical approach.

Overall, these results show that our parametric implementation, when combined with DVS, as seen in ParaScale, outperforms implementations that do not make use of parametric knowledge. The most significant gains in energy conservation are seen when leakage is considered in the processor. Without taking leakage into account, we still obtain solid gains. Thus, we can achieve significant energy savings on contemporary processors, without significant leak-

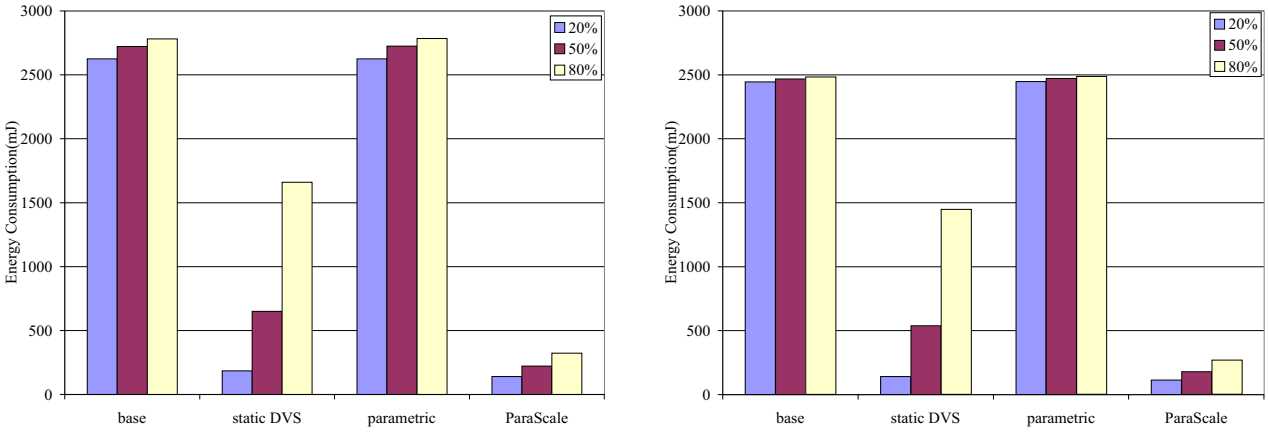
Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)
0-7695-2490-7/05 $20.00 © 2005 IEEE



(a) 2x Overestimation Factor

(b) 10x Overestimation Factor

Figure 6. Energy consumption for PCG Wattch Model



(a) 2x Overestimation Factor

(b) 10x Overestimation Factor

Figure 7. Energy Consumption for PCGL Wattch Model

age as a contributor to power consumption. Any technique that further reduces leakage will inadvertently result in additional gains while using our parametric techniques, *i.e.*, our parametric approach complements leakage awareness.

7. 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 [43]. 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 [33, 36, 19, 25, 21, 32, 45, 15, 28, 24, 13, 29, 41, 40].

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.* [12]. Bernat and Burns proposed algebraic expressions to represent the WCET of programs [7]. 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 [8]. Yet, the combiner functions are not without problems, and timing of basic blocks requires architectural

Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)
0-7695-2490-7/05 $20.00 © 2005 IEEE

knowledge similar to static timing analysis tools.

Parametric timing analysis by Vivancos *et al.* [42] first introduced techniques to handle variable loop bounds as an extension to static timing analysis. Their 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 [38]. 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 larger than 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 [4, 5]. 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.

The most closely related work in terms of intra-task DVS is the idea of power management points (PMPs) [2, 3, 1]. 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 [9]. Again, our results could potentially be further improved by also benefiting from knowledge about past execution, which may lead to additional power savings. This is subject to future work.

8. Conclusion

In this paper, we have shown how parametric timing analysis can benefit decision-making processes during online scheduling, particularly for power-aware scheduling. The technical contributions are as follows. First, a fixed-point approach for embedding parametric formulae into application code is developed, which bounds the worst-case execution time of not only the application code but also the embedded parametric functions and their calls. Prior to entering a parametric loop, the actual loop bounds are discovered and can thus provide a lower WCET bound for the remaining execution of the task. Second, we quantify the benefits from parametric analysis in terms of power savings for sole intra-task DVS as well as ParaScale, our combined intra-task and static inter-task DVS. Here, parametric loops are exploited to gradually scale down the frequency as parametric loop bounds are discovered one by one. We observe savings ranging from 66% to 80% in power over DVS-oblivious techniques, depending on system utilization and the amount of overestimation for loop bounds. These benefits are unique to parametric timing analysis and cannot be achieved by conventional timing analysis methods due to lack of knowledge about remaining execution times.

References

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

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

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

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

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

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

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

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

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

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

Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)
0-7695-2490-7/05 $20.00 © 2005 IEEE

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

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

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

[14] J. K. D. Shin and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. In *IEEE Design and Test of Computers*, March 2001.

[15] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2/3):131–181, Nov. 1999.

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

[17] F. Gruian. Hard real-time scheduling for low energy using stochastic data and dvs processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01*, Aug 2001.

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

[19] M. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *IEEE Real-Time Systems Symposium*, pages 68–77, Dec. 1992.

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

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

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

[23] Y.-H. Lee and C. M. Krishna. Voltage-clock scaling for low energy consumption in fixed-priority real-time systems. *Real-Time Syst.*, 24(3):303–317, 2003.

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

[25] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, Dec. 1994.

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

[27] Y. Liu and A. K. Mok. 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*, May 2003.

[28] K. Lundqvist and G. Wall. Using object oriented methods in Ada 95 to implement linda. In *Ada Europe*, 1996.

[29] T. Mitra and A. Roychoudhury. A framework to model branch prediction for wcet analysis. In *2nd Workshop on Worst Case Execution Time Analysis (WCET)*, June 2002.

[30] S. Mohan, F. Mueller, D. Whalley, and C. Healy. Timing analysis for sensor network nodes of the atmega processor family. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 405–414, Mar. 2005.

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

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

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

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

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

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

[37] S. Saewong and R. Rajkumar. Practical voltage-scaling for fixed-priority rt-systems. In *Proceedings of the ninth IEEE Real-Time and Embedded Technology and Applications Symposium*, May 2003.

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

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

[40] S. Thesing, J. Souyris, R. Heckmann, F. R. andM. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics. In *Proceedings of the International Performance and Dependability Symposium (IPDS)*, June 2003.

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

[42] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Embedded Systems*, volume 36 of *ACM SIGPLAN Notices*, pages 88–93, Aug. 2001.

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

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

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

[46] F. Zhang and S. T. Chanson. Processor voltage scheduling for real-time tasks with non-preemptable sections. In *IEEE Real-Time Systems Symposium*, Dec. 2002.

Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)
0-7695-2490-7/05 $20.00 © 2005 IEEE