Hybrid Timing Analysis of Modern Processor Pipelines via Hardware/Software Interactions *

Sibin Mohan and Frank Mueller
Dept. of Computer Science, Center for Efficient, Secure and Reliable Computing,
North Carolina State University, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu

Abstract

Embedded systems are often subject to constraints that require determinism to ensure that task deadlines are met. Such systems are referred to as real-time systems. Schedulability analysis provides a firm basis to ensure that tasks meet their deadlines for which knowledge of worst-case execution time (WCET) bounds is a critical piece of information. Static timing analysis techniques are used to derive these WCET bounds. A limiting factor for designing real-time systems is the class of processors that can be used. Typically, modern, complex processor pipelines cannot be used in real-time systems design. Contemporary processors with their advanced architectural features, such as out-of-order execution, branch prediction, speculation, prefetching, etc.*, cannot be statically analyzed to obtain* **tight** *WCET bounds for tasks. This is caused by the non-determinism of these features, which surfaces in full only at runtime.*

In this paper, we introduce a new paradigm to perform timing analysis of tasks for real-time systems running on modern processor architectures. We propose minor enhancements to the processor architecture to enable this process. These features, on interaction with software modules, are able to obtain tight, accurate timing analysis results for modern processors. We also briefly present analysis techniques that, combined with our timing analysis methods, reduce the complexity of worst-case estimations for loops. To the best of our knowledge, this method of constant interactions between hardware and software to calculate WCET bounds for out-of-order processors is the first of its kind.

1. Introduction

Embedded systems are increasingly deployed in safety-critical applications and environments. Examples include avionics, power plants, automobiles, *etc.* The software used, in general, must be validated. This traditionally amounts to checking the correctness of the tasks, and in particular, the input/output relationship. Many embedded systems also impose timing constraints, which, if violated, may result in fallouts that are dangerous to the environment. Such systems are typically referred to as *real-time systems*. They impose timing constraints (termed "deadlines") on computational tasks to ensure that results are available on time. Often, approximate results provided on time are more useful than correct results provided late (past the deadline). One critical piece of information required by real-time systems designers is the worst-case execution time (WCET) for each task. This is used to verify that tasks meet their deadlines.

Static timing analysis [6, 11, 15, 20, 26–28, 30, 36] provides bounds on the WCET of tasks. The *tighter* that these bounds are relative to the actual worst-case times, the greater the value of the analysis. Of course, even tight bounds must be *safe* in that the true WCET must *never* be underestimated; the WCET bound may at most match or otherwise overestimate the true WCET.

A serious handicap in performing static timing analysis is the complexity of modern processors and their functional units. Various features that decrease *average* execution times for tasks are often detrimental for worst-case timing analysis. Out of order (OOO) processing [29] and branch prediction [33] are two important features in modern processors that introduce non-determinism to task execution, which cannot be resolved at compile time [4, 9, 16]. Hence, designers of real-time systems are often forced to use less complicated, older and inherently less powerful processors. In this paper, we attempt to bridge this gap by means of our *CheckerMode* infrastructure, which combines the best features of both, static and dynamic analysis, to create a novel hybrid mechanism for WCET analysis.

We propose minor enhancements to the micro-architecture of future processors that will aid in the process of obtaining accurate WCET bounds. A "checker mode" is added to processors that will, on demand, capture varying levels of information as "*snapshots*" of the processor state. This information is communicated to a software module that stores the various snapshots and also drives the execution of instructions in the processor along statically determined paths. Accurate timing information for each path is then captured. These snapshots are also used to backtrack to an earlier state and then restart along a different path. Execution times obtained for each path are analyzed and then combined by the software driver to calculate an accurate WCET for the entire program/function.

* This work was supported in part by NSF grants CCR-0310860, CCR-0312695 and CNS-0720496.

Decisions on where to obtain snapshots, the level of detail required for each snapshot, *etc.* are made by the software controller ("driver"). Timing results for each straight-line path are then fed back to the software module. The software module (similar to a static/numeric timing analyzer), then combines the timing results for individual paths to obtain a bound on WCET for the entire task. The cache states, the state of the branch predictor, the pipeline, *etc.*, for each of the paths, are also considered while performing these calculations. To time an alternate path, the information from the previous snapshot is restored onto the processor function units to reflect the state of the system when the choice between the paths was made.

The ability to capture these snapshots is disabled during normal execution, so as to not interfere with regular program execution. We evaluate this approach by implementing additional micro-architectural functionality (the ability to capture snapshots, to restore a previous snapshot on to the processor function units and the ability to obtain accurate timing results for parts of the program) on a customized SimpleScalar [8] framework that is configured in a manner similar to modern processor pipelines. We also introduce techniques to reduce the complexity of analysis for loops to ensure that the analysis overhead is independent of the number of loop iterations. To the best of our knowledge, this method of using a hardware/software co-design technique to obtain accurate WCETs for modern out-of-order processors is a first of its kind.

Plausibility of the approach: The proposed hardware enhancements are realistic. The support for speculative execution due to dynamic branch prediction, precise exception handling and precise hardware monitoring, and even most of the internal buffers required by our design already exist in modern high-end embedded processors. For example, the ARM-11 features out-of-order execution, dynamic branch prediction, and precise traps, which requires shadow buffers (for registers, branch history tables *etc.*) [12] in order to recover to a prior execution state. In addition to these features, the Intel x86 architecture supports Precise Event Based Sampling (PEBS) with user access to selected shadow buffers [34]. Future processor extensions also make heavy use of checkpoint buffers [13]. Our proposed design makes such buffers uniformly available to the user. We also propose enhancements to the ALU and branch logic to handle the new semantics for NaN operands required by the CheckerMode (see Section 2), which are minor modifications compared to the space and complexity of the already existing shadow buffers. In fact, most processors already implement a NaN representation for floating point values (and an equivalent bottom value for integers), which is generated when undefined arithmetic (*e.g.*, divide-by-zero) is performed and results in an exception (trap). The sole modification suggested by us would be to gate the exception, *i.e.*, suppress it in CheckerMode, and proceed with arithmetic operations in the presence of NaN values.

The process of timing analysis then amounts to timing sequences of paths by saving and restoring snapshots of processor state in a coordinated fashion. While this process can be lengthy, it still remains independent of the input to the program and can be run overnight, even in a parallelized manner. Since this is an offline task to be performed during system design and validation, the cost is secondary and does not affect the dynamic, run-time behavior of the system. In practice, such a full verification of WCET bounds is generally only warranted after extensive code changes during development and for each software deployment, including system upgrades.

Another shortcoming aspect of static timing analysis approaches developed so far is given by their targeting of a generic processor type based on vendor-supplied design details. In such an approach, each new processor design requires that the timing model be manually adapted while our method automatically adapts with changing processor details. Furthermore, such timing models are only as good as the information provided by the vendor, which may not reveal all details of the design. For example, Intel's CPU stepping index indicates subtle processor modifications within the same CPU family but does not reveal all details. Our CheckerMode infrastructure avoids this detailed level of processor modeling and allows vendors to protect their IP while providing a method to obtain highly accurate timing. In fact, fabrication variability (due to smaller die sizes) in the production processes used these days already result in timing variability between two processors originating from the same batch [7, 22]. Access latencies within a cache may actually *differ* from one line to another. Hence, generic timing analysis of a processor series becomes meaningless in such a setting. Since our approach observes the execution time on an actual processor, such variability is captured.

By introducing the CheckerMode concept, we widen the scope of processors that may be used in a real-time system. Contemporary processors with state-of-the-art functionality and performance may subsequently be used in real-time systems. We believe that this also changes the landscape for timing analysis in that more accurate results can be obtained on modern pipelines without risk of losing functionality. In a world of increasingly specialized components, the idea that some processors could be designed specifically for use in real-time and embedded systems has already caught on, *e.g.*, with designs that customize generic core, such as the ARM-7/9/11 licensed by Qualcomm and many others. This is especially true in the design and testing phases for the real-time systems being created. These processors would not behave any differently during normal execution but would only have the additional characteristic that more information can be gathered from them during the analysis phase. Hence, we can be assured that the additional features will not further complicate the analysis.

This paper is organized as follows. Section 2 introduces the CheckerMode infrastructure. Section 3 provides insights on the techniques used to reduce the complexity and overheads for worst-case analysis of loops. Section 4 explains the experimental setup. Section 5 enumerates the results of our experiments. Section 6 discusses the related work in

timing analysis contrasts our work to prior work. Section 7 talks about future avenues for this work. Section 8 summarizes the contributions.

2. CheckerMode

The *CheckerMode* infrastructure, detailed in this section, provides the means to obtain accurate WCET values for modern processor pipelines. It encompasses enhancements/additions to the microarchitecture while closely interacting with software to obtain WCET bounds. We propose to design embedded processors, that in addition to executing software normally (in a so-called *deployment mode*), are capable of executing in a novel *CheckerMode* that supports timing analysis.

Assumptions: In the following, we constrain our work to address the unpredictable nature of out-of-order instruction execution in contemporary high-end embedded processor pipelines. Other complexities, such as memory hierarchies, including caches, dynamic branch prediction and timing anomalies [24] are beyond the scope of this initial work and will be addressed in the future. Tasks are analyzed in *isolation*. Preemptions and cache-related preemption delays, handled by orthogonal work [31], could be incorporated in the future and should not require any changes to our approach since their analysis occurs at a higher level.

The CheckerMode provides cycle-accurate bounds on the WCET by assessing alternate execution paths in a program. In deployment mode, a processor executes along just one path following a conditional branch; which path is executed may depend on the input data. In CheckerMode, a processor no longer proceeds with conventional data-driven execution. Instead, it executes all alternate paths, one at a time, following each conditional branch in order to find the path with the largest execution time. Before the execution of each alternate path, the original execution context (including caches, branch history tables etc.) is restored to correctly simulate the effect of alternations in isolation from one another. These low-level WCET results are propagated inter-procedurally in a bottom-up fashion (over the combined control-flow and call graphs) until the WCET for an entire task has been computed.

Consider a task that consists of a number of feasible execution paths. The execution times for these paths are obtained by actual execution in CheckerMode through the processor pipeline. The execution time for each path is then captured and stored. When conditional execution arises, all alternate paths are timed separately on the pipeline. The timing information as well as the "state" of the processor (determined by the cache state, branch predictor state, register state, *etc.*) are combined when alternate paths join. The combination is performed such that the state that results from the combination must not underestimate the execution time of the alternate paths or even the future execution of the task. A set of timing schemes for individual paths as well as combinations of paths, derived from this methodology, is discussed in the results section.

Prior to the execution of alternate paths, a "snapshot" of the processor state is obtained and stored. After the execution of one of the alternate paths, its state is recorded for later combination with other paths. Then, the state of the processor is restored to the one that existed before the path started executing. This is achieved by restoring the state (*e.g.*, of each of the parts of the pipeline) from the previously captured snapshot.

Consider the simple control-flow graph (CFG) in Figure 1. The CFG contains two possible paths – if the branch is taken, it follows path $1 \rightarrow 2 \rightarrow 4$; if it is not taken, it follows path $1 \rightarrow 3 \rightarrow 4$. When CheckerMode execution reaches basic block 1, a snapshot (snapshot 0) of the processor state is captured and stored. The amount of information to be captured can vary depending on the type of analysis required and can be made configurable. Execution then proceeds down one side of the CFG – say, the taken path. When execution of the path is complete, at basic block 4, another snapshot (snapshot 1) of the processor state is captured and stored. The time taken to execute this path is also measured and sent to the timing analyzer. The program counter is then reset to basic block 1 (the branch condition) to trace execution down the other side (not-taken) and to subsequently capture the execution time for that path. Before execution proceeds along the not-taken path, the state of the processor is *restored* to the previously saved snapshot (snapshot 0). This isolates the effects of execution of one path from that of another. Once the processor state from snapshot 0 is written back, execution from basic block 1 proceeds down the not-taken path ($1 \rightarrow 3 \rightarrow 4$) before the processor state (snapshot 2) and execution time are captured once again. Only then can the CheckerMode unit shift its focus to the code that follows basic block 4. For execution to proceed from basic block 4, the processor must be set to a consistent state. At this point, we perform a *merge* of the snapshots from the two paths. The merge must be performed such that the worst-case behavior of the subsequent code is preserved. Hence, we must merge the state of all processor units captured in preceding snapshots. Once a merge has been performed, the new state must be written back to the processor and execution continues from that point on.

The hardware-supported CheckerMode is complemented by software analysis to govern checker execution (see Figure 2). The *analysis controller (or driver)* steers checker execution along distinct execution paths, *i.e.*, it indicates which direction a branch along the path should take till all paths have been traversed. The timing information and the

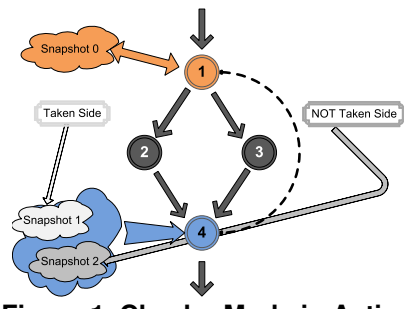


Figure 1. CheckerMode in Action

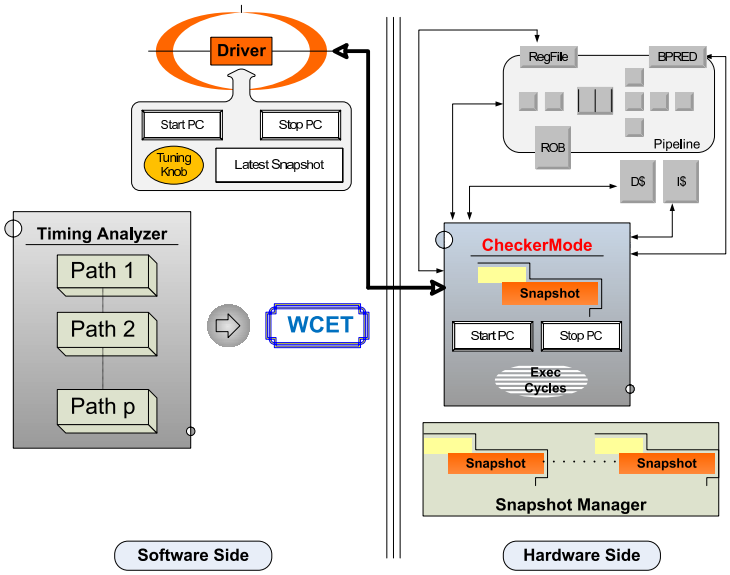


Figure 2. CheckerMode Design

states of the processor obtained for each possible path are then used by a "timing analyzer" to obtain the WCET for the entire task (or even certain code sections). Each of these is explained in the following sections.

2.1. Processor Enhancements

In our work, the embedded hardware is enhanced to support explicit access to the unit-level context of hardware resources, which can be saved and restored. The analysis phase restores a context prior to examining a path and then saves the newly composed context at the end of a path, together with the timing for the path.

Hence, the novel **CheckerMode** unit of the processor supports the following functions:

(a) Capture *snapshots* of the processor state and communicate them to the software controller. A snapshot captures the current state of the processor pipeline, associated functional units and caches, ROB, etc.

(b) Reset the processor to a previously saved state. Given an earlier snapshot, the state of the processor pipeline, caches, functional units, *etc.*, is overwritten with information from the stored snapshot.

(c) Start and stop execution between any two program counter (PC) values. This includes support to calculate the number of cycles elapsed between the execution of the given start and stop PCs.

The right-hand side of Figure 2 shows the details of the hardware side of the design. The CheckerMode unit must be able to read and write to the various functional units of the processor. The CheckerMode unit is controlled by the driver (or controller) on the software side.

2.2. Software Overview

The left-hand side of Figure 2 illustrates the various components that make up the software side of the design. It consists of the following components:

Timing Analyzer (TA): The TA breaks down the task code into a control-flow graph (CFG) and then extracts path information from it. Using this information, the TA is able to determine the start of alternate execution flows – points where snapshots must be obtained. It also provides the start and stop PCs to the driver and obtains the WCET and processor state for that particular path from the driver.

Snapshot Manager (SM): The SM maintains various snapshots that have been captured as well as the PCs at which they were obtained. SM abstractions can be integrated into the processor as depicted in Fig. 2, or, alternately, into the driver within the software controller.

Driver: The driver controls the hardware side of the system. It instructs the hardware on when to start and stop execution, when snapshots must be captured, and when the state of the processor must be reset to a previous snapshot, as detailed below.

The input to the TA is the executable of a task. Assembly information is extracted (with PCs) from an executable and then converted to internal representations as combined control-flow and call graphs. The start and stop PCs provided by the TA encapsulate a single path. The TA, the driver, and the SM interact to decide which snapshot corresponds to which path, which PC, *etc.*, and thereby control program execution.

The TA is responsible for obtaining the final WCET for the entire program as well as various program segments (functions/scopes). It "combines" the information from various paths (execution time, pipeline state, *etc.*) for this purpose. The driver, also part of the software system, is described in more detail below.

2.3. Driver/Analysis Controller and Tuning

The driver is responsible for controlling processor operations. Besides directing the execution of the code on the pipeline, it relays instructions from the TA such as when to capture/restore snapshots. The driver represents the interface between the hardware and software components of the CheckerMode design. The driver contains information about the start and stop PCs that define the start/end points of the path to be timed. It also stores the latest captured

snapshot. The driver maintains information about which instruction is a branch and where snapshots need to be captured. It also relays information in the other direction – from the hardware to the timing analyzer – *e.g.*, the path execution time.

2.4. False Path Identification and Handling

A principal component of the analysis controller is a queue of saved processor contexts guiding path exploration. In some cases, not all paths need to be considered, as implied by these contexts. For example, a path can be dropped if static analysis concludes that this execution path cannot be executed (*i.e.*, it is a "false path"). Similarly, if a path can be shown to be shorter than some other paths that have already been explored, then again this path can be dropped from the queue.

2.5. Analysis Overhead

We can reduce the complexity of determining the WCET by *partial execution of loops* such that the analysis overhead is independent of the number of loop iterations. Using our prior approach of a fixpoint algorithm to determine a stable execution time for the loop body [3], we can steer loop executions such that paths of a loop body are repeatedly executed till a stable value is reached. The controller records the decaying execution times for each iteration, up to the fixpoint, using the CheckerMode hardware. When reaching the fixpoint, the WCET of the remaining loop iterations up to the loop bound is calculated by a closed formula based on the fixpoint value. Typically, loops reach a fixpoint after only 2–4 iterations, which implies that this partial execution can reduce the overhead of WCET analysis significantly. Thus, the complexity of WCET analysis is *independent of the number of iterations*, *i.e.*, it does not depend on the actual execution time of analyzed code.

2.6. Input Dependencies

In CheckerMode, input-dependent register values are deemed *unknown*, which is internally represented in a manner similar to NaN (not-a-number) values already existing in floating point units (and similarly for integer ALUs). Operations on unknown values are straightforward: if *any* input is unknown then the output is also unknown. It is necessary to represent the known/unknown status of condition codes at the bit level. A branch condition based on an unknown value then indicates a need to consider alternate paths. Conversely, concrete (known) values are evaluated as always and input-invariant branches will result in timing of only the taken execution path.

We alter the semantics of execution in CheckerMode to include this NaN value. *E.g.*, addition will now be rewritten as:

$$r_{result} = \begin{cases} \mathbf{NaN} & if\ r_a = \mathbf{NaN} \bigvee r_b = \mathbf{NaN} \\ r_a + r_b & otherwise \end{cases}$$

Hence, any operation with NaN as one of the operands will result in NaN (unless the result is independent of that particular operand, *e.g.*, multiplication with 0 will always result in 0). We developed similar enhancements for other instructions that depend on input-dependent or memory-loaded operands.

Benchmark	Function
ADPCM	adaptive pulse code modulation
CNT	Sum and count of positive and negative numbers in an array.
FFT	Finite Fourier Transform
LMS	Least Mean Square Filter
MM	Matrix Multiplication
SRT	Implementation of Bubble Sort.

Table 1. Subset of C-Lab Benchmarks

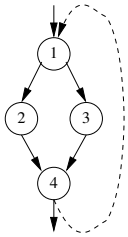
3. Reduction of Analysis Overhead for Loops

Analysis of loops, especially static analysis, increases the complexity of our analysis for various reasons. All iterations of the loop may have to be enumerated or symbolically executed to determine the worst-case execution bounds for the entire loop. This is not always a trivial task. Further complexities arise if the loop body consists of multiple alternating paths. Also, actual execution bounds for the loop may not be known statically due to input dependencies thus preventing us from determining the actual execution bounds for the loop.

The complexity of loop analysis is reduced by *partial execution of loops* so that the analysis overhead is independent of the number of loop iterations. Based on prior work that utilizes a fixed-point algorithm to determine stable execution times for loop bodies [3], we steer loop executions such that the paths in a loop body are executed repeatedly till a stable value is reached. A controller records the monotonically decreasing execution times for each iteration (up to the fixed point). Once the fixed point is reached, the WCET of the remaining loop iterations (up to the loop bound) is calculated using a closed formula. Experimental results in section 5 indicate that loops reach a fixed point after only 3 iterations to account for pipeline effects, and, another two iterations are required on average in the presence of caches. The details are beyond the scope of this paper (see [25]). This implies that this technique of partial executions can significantly reduce the overhead of WCET analysis. Thus, the complexity of WCET analysis is *independent of the number of iterations*. It does not depend on the dynamic execution time of the analyzed code.

4. Experimental Framework

We have implemented the key components of our design in the SimpleScalar processor simulator [8]. This cycle-accurate simulator can be configured for the various processor and branch prediction schemes.



(a) CFG

Path	Cycles		
	SimIO	SupIO	OOO
bb 1	36	20	20
bb 2	8	4	20
bb 3	38	13	29
bb 4	15	5	22
bb 4'	15	5	16

(b) Cycles (Short Technique)

Figure 3. Benchmark and Measured Cycles

Path	SimIO	delta	SupIO	delta	OOO	delta
BB1	82	BB1-BB0=56	66	BB1-BB 0=4	47	BB1-BB0=1
BB1,2	114	BB1,2-BB1=32	94	BB1,2-BB1=28	59	BB1,2-BB1=12
BB1,3	241	BB1,3-BB1=159	131	BB1,3-BB1=65	92	BB1,3-BB1=45
BB1,2,4	151	BB1,2,4-BB1,2=37	97	BB1,2,4-BB1,2=3	61	BB1,2,4-BB2=2
BB1,3,4	278	BB1,3,4-BB1,3=37	134	BB1,3,4-BB1,3=3	94	BB1,3,4-BB1,3=2

Figure 4. Measured Cycles (Aggregate Technique) for Synthetic Benchmark

We used SimpleScalar in three configurations:
(1) *Simple-IO (SimIO)* simulates a simple, in-order (IO) processor pipeline (pipeline width 1, instruction issue in program order);
(2) *Superscalar-IO(SupIO)* with a pipeline width (from fetch to retire) of 16 and in-order instruction execution; and
(3) *Out-of-order (OOO)* execution with the same pipeline width as in Superscalar-IO.

We used the C-Lab benchmarks [10] enumerated in Table 1 for our experiments. We also conducted experiments on a synthetic benchmark whose control-flow structure is depicted in Figure 3(a).

Execution time for paths is measured using four different techniques, extending from the use of basic blocks (BB) [1] to paths (sequences of consecutive BBs):
(1) *Short* measures the execution time of a single BB, starting from the time that **any** instruction in the BB/path moves into the *execute* stage of the pipeline and finishing when the last instruction of the BB/path exits from the *retire* stage.
(2) *Path-Short* captures the execution time for paths (concatenated BBs) using the "short" technique so that timing starts at the first BB and ends with the last BB in the path.
(3) *Path-Aggregate* captures the time for concatenated paths so that timing starts at the first BB of the first path and ends with the last BB of the last path.
(4) *Program-Aggregate* includes the time from the start of the execution (main function) to the end of a BB in the path being timed, starting when the first instruction in the main function is *fetched* and finishing when the last of the path exits from the *retire* stage.

5. Results

The results obtained for the "short" technique (Table 3(b)) show that timings for the processor modes SimIO and SupIO accurately reflect the actual WCET bounds, both for single BBs and paths. However, the OOO results exceed those of SupIO, due to early out-of-order execution of some instructions in parallel with other instructions from prior BBs in the path. Timing is started when any instruction in the relevant path comes into the execute stage of the pipeline, which could very well happen even when the previous path is not complete due to the inherent nature of out-of-order execution. Since timing only stops when the last instruction in the current path retires, the total execution time includes some time from execution of instruction in the previous path. Hence, the observed execution time includes cycles for instructions from earlier paths, which were not supposed to be timed. Even timing multiple BBs of a path in sequence ("path-short" technique) does not alleviate this problem. bb4 and bb4' represent the same code – the difference is the path taken to get to basic block 4. In the first case, the "then" case of the branch was selected and in the second case, then "else" case was followed.

Benchmark	SimIO	SupIO	% Savings	OOO	% Savings
ADPCM	1340	486	63.7	367	72.6
CNT	356	197	44.6	76	78.7
FFT	1047	439	58.1	288	72.5
LMS	839	457	45.6	236	71.9
MM	161	144	10.6	58	64.0
SRT	330.2	198	40.1	93	71.8

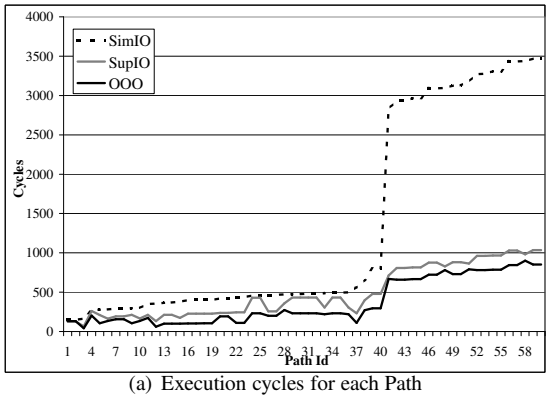
Table 2. Averaged WCECs for C-Lab Benchmarks

In contrast, the "aggregate" technique (Figure 4) reflects the time from instruction fetch (instead of execute) and also times longer paths. This addresses the above problem of early execution by some instructions because in the long run, timing longer paths reduces the inaccuracies from interactions between individual instructions . Results show a strict ordering of execution cycles for $SimIO \geq SupIO \geq OOO$, as expected by the amount of instruction parallelism, since time is measured from the first fetch of an instruction. The differences between paths ("delta") provide a bound on the number of cycles for the tail BB in the path, thus excluding any pipeline overlap with prior BBs. Hence, these delta values can be used to assess the amount of cycles attributed to specific BBs alone. They also adhere to the same strict ordering. In general, such timing results are only valid in the same execution context/path, *i.e.*, different BB sequences of one path may influence subsequent BBs in the control flow.

5.1. C-Lab Benchmark Results

We extracted all paths from each of the C-lab benchmarks and then timed them independently using our CheckerMode framework in each of the three configurations (SimIO, SupIO and OOO). Figures 5 and 6 summarize our results for the ADPCM, LMS and SRT benchmarks, respectively. ADPCM is the largest benchmark in the C-lab suite, with 14 functions and 60 paths, while LMS and SRT are smaller benchmarks with 10 paths each. Results are sorted in ascending order based on the timing results for the SimIO configuration. From all three graphs, we see the $SimIO \geq SupIO \geq OOO$ ordering except for one path in the SRT benchmark, which we will explain later.

Figure 5(a) shows the timing results for the ADPCM benchmark, while Table 5(b) lists the various functions in ADPCM as well as the number of instructions and paths in each function. These results show the strict ordering for the

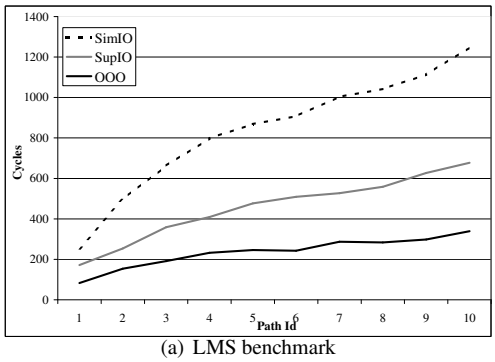


(a) Execution cycles for each Path

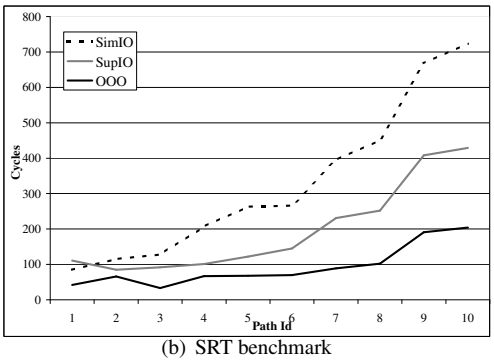
Function	Number of Instructions	Number of Paths
abs	18	2
filtep	35	1
logsch	36	2
logscl	37	2
filtez	48	2
uppol1	49	8
uppol2	58	8
quantl	65	6
main	88	4
upzero	122	5
decode	317	4
encode	330	16

(b) Number of Instructions and Paths for each function in ADPCM benchmark

Figure 5. Timing Results for the ADPCM Benchmark



(a) LMS benchmark



(b) SRT benchmark

Figure 6. Measured execution cycles for C-Lab Benchmarks

three configurations, with SimIO results being the largest and OOO being the smallest. From the same graph, we see that the timing results for SimIO increase significantly around path 42. This is because paths 42 – 61 originate from the "encode" and "decode" functions of the ADPCM benchmark and contain a larger number of instructions and, in the case of *encode*, a large number of paths as well. While there is enough parallelism in the code for SupIO and OOO to exploit, the SimIO configuration, with its in-order behavior and single width pipeline, is unable to scale as well as the other two configurations. This also shows that the number of dependencies between instructions in the two functions is not very high, as OOO is able to scale well to handle the larger instruction load.

The graph for LMS (Figure 6(a)) shows that all three configurations scale in a similar fashion for larger paths. It is interesting to note that the timing results for SupIO are approximately half of that for SimIO. Similarly, the timing results for OOO are approximately half that of SupIO. We see similar results for the SRT benchmark as well (Figure 6(b)), except for the shortest path (path 1). This path is so short that the effects we described at the beginning of Section 5 become apparent – *i.e.*, timing is started when the first instruction of the program is fetched and stopped when the final instruction is retired. Hence, the first instruction has to wait for a while before it is dispatched. When the paths are very short, the pipeline contains a large number of instructions that do not belong to the particular path being timed, hence bloating the results for pipelines with larger width. The single width IO configuration does not suffer from this problem as the instruction is dispatched immediately after being fetched.

The FFT and MM benchmarks also show similar results. The results of all six benchmarks are summarized in Table 2. The second, third and fifth columns are the *worst-case* number of cycles for each benchmark averaged across all paths. The fourth and the sixth columns show the average savings for each benchmark for the preceding configuration (preceding row in the table) as compared to SimIO (column 2). Specifically, the fourth column shows the average savings for SupIO over SimIO, and the sixth column shows the average savings for OOO over SimIO. These savings are based on the average across all paths.

5.2. Partial Analysis of Loops

Our objective is to leverage path timings under the "program-aggregate" technique as a refinement to the "aggregate" technique discussed so far. Consider the construct depicted in Figure 3(a) embedded within the loop (dashed vertex). We must assess consecutive executions of paths within the loop. For *E.g.*, within one iteration, the L-left (BB 1,2,4) and R-right (BB 1,3,4) paths are timed; within two iterations, concatenations of all permutations for these paths

are timed (L-L/L-R/R-L/R-R); and so on for three and four iterations. Since this search space grows exponentially with the number of alternate paths and loop iterations, we propose to devise a bounded technique to limit the path space in depth and breadth.

Table 3 depicts the results for 3 iterations of this loop around the left (L) and right (R) paths for the 3 processor models. It also distinguishes path composition without overlap ($+$) and with overlap (o), where the former is equivalent to draining the pipeline while the latter captures continuous execution. The difference between the compositions is depicted as δ. The table depicts constant δ values for all processor models regardless of the paths being executed (D-caches are disabled here.) More significantly, early results from our experimentation environment indicate that three iterations are sufficient to reach a fixed point for the δ values while considering only pipeline effects. Beyond that point, concatenation of another iteration results in a constant increase in cycles for this path, and this behavior does not change for the remainder of the loop. For instance, a 2-path experiment (omitted here) resulted in exactly half the δ values of the 3-path experiment, which reinforces the claim about reaching a fixed point.

Tables 4 and 5 depict the two and three level compositions for the FFT benchmark. The first column depicts the loop ID while the second column shows the particular combination of paths being timed. For example, "0_1_0" represents a three-way combination of paths "0", "1" and "0". As before, the "+' ' represents path timing without overlap while the "o" represents path times with overlap. These tables indicate that the loops reach a fixed point within 2-3 iterations. For example, the δ values for the "0_0" path combination of loop 1 from Table 4 is exactly *half* that of the corresponding δ values for the "0_0_0" combination of the same loop (Table 5). More detailed loop analyses are presented elsewhere [25]. Other C-Lab benchmarks show similar results. They are omitted here due to space considerations.

We exploit this behavior to limit the search depth. In addition, we limit the breadth of the search by limiting timings to k consecutive splits in the control flow; this bounds the growth due to consecutive conditionals (alternating joins and splits in the control flow). Such restrictions may result in pessimism when path overlap can only be loosely bounded (instead of timing it in the CheckerMode).

Path	SimIO			SupIO			OOO		
	$+$	o	δ	$+$	o	δ	$+$	o	δ
LLL	453	443	10	291	193	98	183	123	60
LLR	580	570	10	328	230	98	216	156	60
LRL	580	570	10	328	230	98	216	156	60
LRR	707	697	10	365	267	98	249	189	60
RLL	580	570	10	328	230	98	216	156	60
RLR	707	697	10	365	267	98	249	189	60
RRL	707	697	10	365	267	98	216	189	60
RRR	834	824	10	402	304	98	282	222	60

Table 3. Path-Aggregate Cycles (3 Iterations) for the Synthetic Benchmark

Id	Path	SimIO			SupIO			OOO		
		$+$	o	δ	$+$	o	δ	$+$	o	δ
0	0_0	1684	1646	38	668	628	40	594	529	65
1	0_0	574	536	38	420	312	108	226	193	33
1	0_1	1268	1231	37	661	553	108	423	390	33
1	1_0	1268	1231	37	661	565	96	423	390	33
1	1_1	1962	1924	38	902	806	96	620	587	33
2	0_0	1684	1646	38	836	712	124	642	553	89
3	0_0	574	536	38	414	309	105	330	245	85
3	0_1	1268	1231	37	655	550	105	527	442	85
3	1_0	1268	1231	37	655	562	93	527	442	85
3	1_1	1962	1924	38	896	803	93	724	639	85

Table 4. Path-Aggregate Cycles (2 Iterations) for the FFT benchmark

6. Related Work

Knowledge of worst-case execution times (WCETs) is necessary for most hard real-time systems. The WCET must be known or safely bound *a priori* so that the feasibility of scheduling task sets in the system may be determined based on a scheduling policy (*e.g.*, rate-monotone or earliest-deadline-first scheduling [23]). Methods to obtain upper bounds on execution time range from dynamic (but unsafe) observation to static analysis (safe but not always tight) [38]. Past work mainly focuses on static timing analysis techniques [6, 11, 15, 20, 26–28, 30, 36].

Recently, hybrid methods have been proposed [6, 17] as well as hardware-related methods [2, 24]. Yet, none of these approaches capture advanced hardware features transpar-

Id	Path	SimIO			SupIO			OOO		
		$+$	o	δ	$+$	o	δ	$+$	o	δ
0	0_0_0	2526	2450	76	1002	922	80	891	761	130
1	0_0_0	861	785	76	630	414	216	339	273	66
1	0_0_1	1555	1481	74	871	655	216	536	470	66
1	0_1_0	1555	1480	75	871	667	204	536	470	66
1	0_1_1	2249	2174	75	1112	908	204	733	667	66
1	1_0_0	1555	1480	75	871	667	204	536	470	66
1	1_0_1	2249	2174	75	1112	908	204	733	667	66
1	1_1_0	2249	2175	74	1112	920	192	733	667	66
1	1_1_1	2943	2867	76	1353	1161	192	930	864	66
2	0_0_0	2526	2450	76	1254	1006	248	963	785	178
3	0_0_0	861	785	76	621	411	210	495	325	170
3	0_0_1	1555	1481	74	862	652	210	692	522	170
3	0_1_0	1555	1480	75	862	664	198	692	522	170
3	0_1_1	2249	2174	75	1103	905	198	889	719	170
3	1_0_0	1555	1480	75	862	664	198	692	522	170
3	1_0_1	2249	2174	75	1103	905	198	889	719	170
3	1_1_0	2249	2175	74	1103	917	186	889	719	170
3	1_1_1	2943	2867	76	1344	1158	186	1086	916	170

Table 5. Path-Aggregate Cycles (3 Iterations) for the FFT benchmark

ently while providing tight bounds. Our work fills this gap and contributes to high confidence in embedded systems design for time-critical missions. While static timing analysis methods, such as our past work [18, 26–28] or abstract interpretation methods [35, 36], can provide reasonably tight bounds for branches that can be statically analyzed, they are not able to provide tight bounds for execution along speculatively predicted branch directions at runtime or out-of-order instruction-issue pipelines. The complexity and overhead of modeling the behavior of even moderately complex pipelines [19] and interactions of instructions within them is high for any of these methods. As detailed in the introduction, each new processor design requires that the model be manually adapted and cannot reflect fabrication-level timing variability within a processor batch. Our method, in contrast, automatically adapts with changing processor details, including timing variations due to fabrication.

Our *CheckerMode* is related to two prior approaches. First, Bernat *et. al.* used probabilistic approaches to express execution bounds down to the granularity of basic blocks, which could then be composed to form larger program segments [6]. Second, the VISA framework [2] suggested architectural enhancements to gauge the progress of execution by sub-task partitioning and exploiting intra-task slack with DVS techniques. Our work combines the benefits of these two prior approaches without their shortcomings. While performing analysis on paths, cycles are measured in a special execution mode of the processor that supports checkpoint/restart and unknown value execution semantics to reflect proper architectural state and path coverage. While Bernat struggled with considerable timing perturbation from instrumentation, the CheckerMode is much less intrusive. Instead of a VISA-like *virtual processor* around a complex core, we promote the CheckerMode as a realistic feature building on existing internal processor buffers widely used for speculation / precise event handling. Hence, our method is able to provide more precise results compared to Bernat's work. In contrast to VISA, is able to support hybrid timing analysis on the actual processor core.

Lundqvist et al. [24] use symbolic execution with a tight integration of path analysis and timing analysis to obtain accurate WCET estimates. They use the concept of an *"unknown"* value to account for register values and addresses that cannot be statically determined, just as the CheckerMode does. However, their work did not utilize a fixed-point approach but rather required each iteration of a loop to be symbolically executed. Furthermore, they did not propose any architectural modification, it focused on static timing analysis over the entire program within an architectural simulator using in-order execution without dynamic branch prediction etc. The term "timing anomaly", *i.e.*, an anomaly in the execution of code in dynamically scheduled processors, stems from their work. It was later generalized by others [5, 24, 32]. Anomalies denote counter-intuitive results in timings, *e.g.*, a cache hit may result in longer execution times than a miss for a given path due to overlapped structural resource conflicts. We contend that the instruction window may be large enough that even if instructions get blocked due to anomalies, other instructions, which are ready, may execute, thus reducing the overall execution time of the program. Thus, by taking a larger context (path) into account, we will provably compensate for localized anomalies at a larger scale within the CheckerMode.

Some early work has suggested probabilistic analysis [6, 14, 21, 37] for handling WCET variations due to software factors (such as data dependency and history dependency). However, these prior approaches for statistical WCET analysis did not model hardware execution time variations caused by process variations.

Our approach with the CheckerMode combines the best features of static and dynamic analysis required for obtaining WCET bounds for modern processors. Section 2 introduced our *hybrid* timing analysis technique that obtains actual execution times for short paths on the actual hardware and then combines these intermediate worst-case bounds, offline, using a static tool.

7. Future Work

We intend to focus on out-of-order processor pipelines and branch prediction. Although we already capture some processor state as "snapshots", we intend to find good solutions to capture state for the out-of-order pipeline and branch predictor features. We will further investigate methods to merge timing and state information for alternate paths. We also intend to study on how to copy state information from a previously captured/newly merged snapshot back onto the processor.

8. Conclusion

We have outlined a "*hybrid*" mechanism for performing timing analysis that utilizes interactions between hardware and software. The *CheckerMode* concept provides the foundation to make contemporary processors predictable and analyzable so that they may be safely be used in real-time systems. Current trends in microprocessor features indicate that our proposed hardware modifications are realistic [34]. Once fully implemented within the SimpleScalar simulator, the CheckerMode unit will have the ability to not only drive execution along given program paths but also to capture and write back processor state to/from snapshots. This will enable us to accurately gauge the execution time for a given program path. We believe this work will enhance the design choices available to real-time systems engineers. The CheckerMode concept will provide them with the ability to use current and future state-of-the-art microprocessors in their systems and utilize a hybrid of static and dynamic timing techniques to validate WCETs.

References

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

[2] 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.

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

[4] I. Bate and R. Reutemann. Worst-case execution time analysis for dynamic branch predictors. In *Euromicro Conference on Real-Time Systems*, pages 215–222, 2004.

[5] C. Berg. PLRU cache domino effects. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

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

[7] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov/Dec 2005.

[8] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin, Madison, July 1996.

[9] C. Burguiere and C. Rochange. A contribution to branch prediction modeling in wcet analysis. In *Design, Automation and Test in Europe*, pages 612–617, 2005.

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

[11] K. Chen, S. Malik, and D. I. August. Retargetable static timing analysis for embedded software. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, October 2001.

[12] D. Cormie. The ARM11 microarchitecture. 2002.

[13] A. Cristal, O. Santana, M. Valero, and J. Martinez. Toward kilo-instruction processors. *ACM Trans. Archit. Code Optim.*, 1(4):389–417, 2004.

[14] S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *22nd IEEE Real-Time Systems Symposium*, pages 215–224, 2001.

[15] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Dept. of Information Technology, Uppsala University, 2002.

[16] J. Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *IEEE Real-Time Embedded Technology and Applications Symposium*, page 152, 2003.

[17] A. Hamann, M. Jersak, K. Richter, and R. Ernst. Design space exploration and system optimization with symta/s - symbolic timing analysis for systems. In *IEEE Real-Time Systems Symposium*, pages 469–478, Dec. 2004.

[18] 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.

[19] R. Heckmann, M. Langenback, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, pages 1038–1054, July 2003.

[20] A. Hergenhan and W. Rosenstiel. Static timing analysis of embedded software on advanced processor architectures. In *DATE*, pages 552–559, 2000.

[21] X. S. Hu, Z. Tao, and E. H. M. Sha. Estimating probabilistic timing performance for real-time embedded systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 9(6):833–844, 2001. 1063-8210.

[22] P. Hurat, Y.-T. Wang, and N. Vergese. Sub-90 nanometer variability is here to stay. *EDA Tech Forum*, 2(3):26–28, Sept. 2005.

[23] 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.

[24] T. Lunqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University, 2002.

[25] S. Mohan and F. Mueller. Fixed-point loop analysis for high-end embedded processor pipelining via hardware/software interactions. In *preparation for LCTES*, page (to be submitted), 2008.

[26] S. Mohan, F. Mueller, W. Hawkins, M. Root, C. Healy, and D. Whalley. Parascale: Expoliting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *IEEE Real-Time Systems Symposium*, pages 233–242, Dec. 2005.

[27] 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.

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

[29] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA*, pages 206–218, 1997.

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

[31] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. In *IEEE Real-Time Systems Symposium*, pages 212–222, Dec. 2006.

[32] J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Universitaet des Saarlandes, 2002.

[33] Smith, J. E. A study of branch prediction strategies. In *Proc. 8^{th} International Symposium on Computer Architecture*, pages 135–148, Minneapolis, 1981.

[34] B. Sprunt. Pentium 4 performance monitoring features. 2002.

[35] H. Theiling and C. Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. In *IEEE Real-Time Systems Symposium*, pages 144–153, Dec. 1998.

[36] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. 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.

[37] G. D. Veciana, M. Jacome, and J.-H. Guo. Assessing probabilistic timing constraints on system performance. *Design Automation for Embedded Systems*, 5(1):61–81, 2000.

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