CheckerCore: Enhancing an FPGA Soft Core to Capture Worst-Case Execution Times *

Jin Ouyang
Dept. of Comp. Sci. & Engr.
The Pennsylvania State University
University Park, PA 16802
jouyang@cse.psu.edu

Raghuveer Raghavendra
Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695
r.raghuveer@ncsu.edu

Sibin Mohan
Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
sibin@cs.uiuc.edu

Tao Zhang
Dept. of Comp. Sci. & Engr.
The Pennsylvania State University
University Park, PA 16802
tzz104@cse.psu.edu

Yuan Xie
Dept. of Comp. Sci. & Engr.
The Pennsylvania State University
University Park, PA 16802
yuanxie@cse.psu.edu

Frank Mueller
Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695
mueller@cs.ncsu.edu

ABSTRACT

Embedded processors have become increasingly complex, resulting in variable execution behavior and reduced timing predictability. On such processors, safe timing specifications expressed as bounds on the worst-case execution time (WCET) are generally too loose due to conservative assumptions about complex architectural features, timing anomalies and programmatic complexities. Hence, exploiting the latest architectures may not be an option for embedded systems with hard real-time constraints where deadline misses cannot be tolerated.

This work addresses these shortcomings by contributing **CheckerCore**. CheckerCore is a mode-enhanced SPARC v8 soft core processor synthesized on an FPGA. During regular execution the core adheres to its original specifications. But when operating in a special time-checking configuration, CheckerCore executes programs irrespective of inputs and steers execution along selected control flow paths. Such execution allows systematic derivation of worst-case execution time (WCET) bounds. This paper presents the design and implementation of CheckerCore and illustrates its use in deriving accurate WCET bounds for a set of embedded benchmarks. Overall, CheckerCore proposes a realistic processor core enhancement that encapsulate processor details without revealing them to users while supporting safe bounding of WCETs. To the best of our knowledge, this is the first contribution of a WCET-enhancing microarchitectural feature besides full processor encapsulations.

*This work is partly supported by NSF grants 0905181, 0905365, 0720659, 0720496, 06-49885, grants from Rockwell Collins and Lockheed Martin, and ONR grant N00014-05-0739.

CASES'09, October 11–16, 2009, Grenoble, France.

Categories and Subject Descriptors

C.3 [**Computer System Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms

Design, Experiment

Keywords

Worst-Case-Execution-Time, WCET, Real-Time, Embedded System, FPGA, SPARC, LEON3, CheckerCore, Shadow Pipeline

1. INTRODUCTION

A key requirement for safety-critical real-time embedded systems is that the timing behavior of software components be *predictable*. In particular, hard real-time systems have timing constraints (deadlines) that, if not met, could results in fallouts that are dangerous to both humans and the environment. Hence, requirements in safety-critical domains (*e.g.* avionics, automobiles, *etc.*) are being extended to require safe and verifiable bounds on execution time. Determining bounds on the *worst-case execution time* (WCET) for embedded software is a critically important problem for next-generation embedded real-time systems.

There exist a variety of techniques to calculate WCET: *(a)* static timing analysis techniques [6, 7] rely on compile-time analysis of task code and derives *safe* WCET bounds that are independent of input values; *(b)* dynamic timing analysis techniques [4, 17] resort to testing and profiling of task code or stochastic methods to estimate WCET values; and *(c) hybrid* techniques [3, 13, 14, 18] use the best of both worlds. While static techniques provide safe WCET bounds, they cannot keep pace with architectural innovations (*e.g.*, out-of-order execution, speculation and dynamic branch prediction) or capture the increasing hardware performance variation due to the fluctuation of manufacturing parameters as processor technology scales. Also, static timing analysis requires detailed simulation of hardware components that is prone to inaccuracy since processor vendors often do not disclose the internal details of their products. Dynamic timing

analysis, on the other hand, has been proved to be unsafe, in that it can underestimate the WCET values [17], which can have dangerous consequences.

Hence, hybrid timing analysis techniques provide a convenient alternative for obtaining accurate WCET bounds for contemporary processors. Of these, the CheckerMode [13, 14] approach is particularly interesting since it provides a mechanism to calculate safe WCET bounds without detailed modeling or even knowledge of the internals of the processor. This approach uses the processor itself as a model and obtains information from it, such as execution time for program paths and "snapshots" of the processor state. While this is quite a novel approach, it is not easy to determine the applicability of these techniques since the original work uses a processor simulator for experiments.

Contributions: This paper contributes a fundamentally new approach to calculating the WCET for real-time programs without requiring detailed models of the hardware. Instead of simulating execution, we promote actual execution in hardware. This not only renders tedious hardware modeling unnecessary but also guarantees correct behavior. It hides architectural complexities and variations in manufacturing technology from end users of the processor. This provides a means to verify bounds on WCET. A further novelty of the approach is that it is demonstrated and evaluated by synthesis of a *SPARC v8 soft core processor* on an *FPGA platform*. This assesses the feasibility of the design and validates a prototype implementation. To this effect, we present a hardware/software hybrid platform to generate WCET estimates of a target processor. Our design is comprised of two main components:
I. a front-end path/timing analyzer running as a program on a host machine and
II. a back-end emulation engine with a native target processor, enhanced with a so-called CheckerMode, synthesized onto an FPGA by morphing a soft core into a *CheckerCore*.

Within CheckerCore, the main function of CheckerMode is to capture execution cycles and other necessary information that represent the worst-case behavior and to communicate it, as *snapshots*, to a front-end software component. The front-end software, in turn, stores and analyzes various snapshots and may "merge" snapshots as required. To enumerate multiple paths, the front-end may also direct the back-end to re-execute a previously captured snapshot on the processor in a manner that preserves the worst-case timing. The final result of the interaction of front- and back-end is a safe bound of the WCET.

We define a command language, in XML, and an API that facilitates the exchange of information between the hardware and software sides. Hence, it is possible to change either the actual softcore (processor) used at the back-end or the timing analysis scheme and yet have the overall system work correctly. We can thus obtain accurate WCET information for different processor types and timing analysis techniques using CheckerCore, as long as both conform to the same API/information exchange mechanism. For this work, we also developed a "front-end" timing analyzer that drives the process of requesting and reasoning about information from CheckerCore.

With the definition of a common interface, manufacturers of embedded processors can implement and provide instruments proposed in this paper as long as it conforms to the interface. Hence, we propose to include CheckerCore, by default, in the processor. This is the major difference between our approach and previous simulation-based approaches. Manufacturers of embedded processors can deliver evaluation versions of processors that include CheckerCores, either as a part of silicon or as encrypted softcores on FPGAs. In addition, the proposed CheckerCore implementation incorporates modular add-ons, such as the shadow pipeline, synchronization logic, *etc.* (Section 5), that are light-weighted (Section 7.3) and can be decoupled. Hence, manufacturers can even deliver commodity CheckerCores to users. The main advantage of this approach is that is avoids the requirement of revealing microarchitectural details while still providing high fidelity. Other benefits are discussed in Section 9.

The rest of the paper is organized as follows. Section 2 lists the assumptions for our work. Section 3 provides an overview of the framework. Section 4 specifies the information captured in a snapshot, and how snapshots are merged to preserve worst-case timing. Section 5 and 6 describe our back- and front-end designs, respectively. Section 7 discusses the experimental result and Section 8 reviews related work. Section 9 summarizes the contributions and further discusses the benefits of CheckerCore.

2. ASSUMPTIONS

This paper restricts itself to analyzing instruction execution in contemporary, high-end embedded processor pipelines. Other complexities (memory hierarchies, including caches, dynamic branch prediction, *etc.*) exist but are subject to future research. We assume that tasks execute in isolation and preemption delays (including cache related preemption delays) are orthogonal to this work. Existing [15] and future techniques to handle these aspects can be incorporated into our framework with minimal changes for tighter results.

Since we only consider the *worst-case* effects of instructions executing through the pipeline, the only *timing* factor that needs to be accounted for is the execution time for the various *paths* that make up the task. Data-dependent instructions may also affect the execution times for each path and, hence, the worst-case time for the entire task. For such an instruction, input values are either available at compile time or the instruction is assumed to execute at its worst-case timing behavior (largest number of cycles).

Note: The timing analysis process in our framework amounts to timing sequences of paths coupled with saving/restoring processor context. It is independent of program inputs and is performed as an **offline** task during the design and/or validation stages. Cost is secondary as the timing process may be performed in the background overnight, *i.e.*, this process *does not* affect the actual runtime behavior of the embedded system on deployment. Such extensive verification, in practice, is typical for initial deployment, for hardware reconfiguration/upgrades or after extensive code changes (development, upgrades, *etc.*).

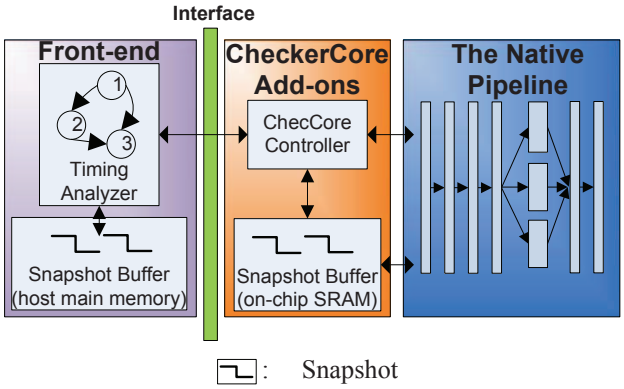

Figure 1: CheckerCore Framework

3. CHECKERCORE

The principle parts of CheckerCore are a front-end component, also referred to as timing analyzer (TA), and the back-end component, CheckerCore. The TA is responsible for deriving the WCET of a program. It uses the program disassembly to construct a control flow graph (CFG). The CFG is then used to drive the back-end to measure the timing of all paths of the program[1]. These timing values are combined to find the longest path(s) in the program and then derive the overall WCET. The back-end is a physical processor with support for a *CheckerMode* whose purpose is to support checkpointing and restarting executions while preserving timing. In this paper, such a mode was implemented within CheckerCore, an extension to an FPGA soft core.

Before alternate paths execute, the original processor context are restored to correctly simulate the effect of execution of alternate paths in isolation from each other. The front- and back-ends interact through an interface that allows the former to steer execution while the latter executes selected code sections along triggered paths in a program's control flow. The interface and the TA are discussed in detail in Section 6. The interaction between front-end and CheckerCore is depicted in Figure 1. The TA interacts with CheckerCore through the CheckerCore controller. It sends commands to start execution from a particular snapshot, to obtain snapshots at various points in the program and to merge snapshots. While the theoretical definition of a *snapshot* is defined in literature [14], a hardware implementation requires certain mechanisms and exposes tradeoffs. We discuss these details as well as techniques on how to capture, restore and merge snapshots in hardware implementations in Sections 4 and 5. The controller subsequently interprets the commands and drives CheckerCore execution appropriately. Snapshots and timing measurements are passed back to the TA. The TA uses the timing information to systematically derive the overall WCET.

4. PRESERVING TIME IN A CHECKERCORE SNAPSHOT

Our theoretical basis for preserving worst-case timing in snapshots is derived from the literature [13, 14], where a proof of correctness for preserving WCET bounds is given. In this section, we only briefly reiterate the key ideas before discussing a number of significant enhancements and complementary changes in this paper that conform with the correctness proof.

[1]A "path" is defined as a sequence of basic blocks that are contiguous w.r.t. execution flow, delineated by changes in control flow.

4.1 Pathological Timing Behavior

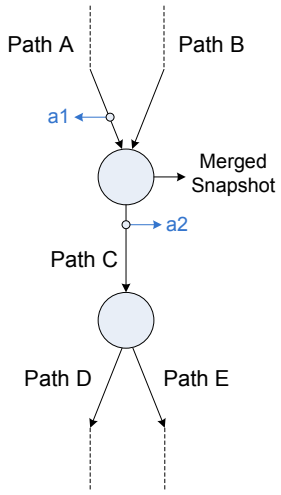

Figure 2: An excerpt of a control flow graph

One simple way to enable rewinding and replaying the execution of a program is to capture all state information in the pipeline, *e.g.*, PC, register file, pipeline registers. Although this will guarantee that the pipeline can be restarted in a functionally consistent state, it does not necessarily preserve the worst-case timing. In addition, only some critical information is related to worst-case timing and only this limited amount of pipeline information needs to be preserved. A simple example in Figure 2 can be used to illustrate on how to determine what pieces of information are to be captured in a snapshot. Among the five paths shown in the figure, path A and path B join at a merge point between instructions a_1 and a_2. Suppose that a snapshot is taken at the merge point and that instruction a_1 is included in the snapshot. If instruction a_2 has either a structural dependency or a data dependency on a_1, then the issue time and completion time of a_1 will affect the worst-case latency of path C (and, hence, paths D and E). Depending on a_1's timing behavior, it is possible that $WCET_D < WCET_E$ or $WCET_D > WCET_E$. Hence, to preserve a_1's impact on worst-case timing behavior, we need to record exactly when a_1 begins and completes execution. If the merged snapshot only contains the functional states, then replaying this snapshot, while "functionally correct", will not be safe in terms of worst case timing.

4.2 Information Included in a Snapshot

From the example in the previous section, one can observe that merely capturing the pipeline state as a whole is insufficient to preserve the worst-case timing. In the rest of this section, we will discuss the information necessary to preserve the worst case timing. The discussion is focused on the in-order processor, which serves as our baseline architecture. While in-order processors presents different requirements for

engineering the snapshot, they still share the same principle with the OOO processors [13, 14].

4.2.1 Snapshot for a Single Execution Context

The general idea of making snapshot is to preserve the various dependencies, and, hence, the impact of worst-case timing. The following items are included in the snapshot of a single execution context. We later explain in Section 4.2.2 that our snapshot also needs to handle multiple execution contexts.

Execution time up to current snapshot: This is necessary for the timing analyzer to select the longest path.

Processor State: Pipeline state is necessary to put the processor into a consistent state when replaying a snapshot. Depending on the microarchitecture, the size of this information can vary widely. For the native processor specifically examined in this work, the relevant pipeline state only includes the PC, the processor state register, and the return address.

Dependency information: The data and control dependency information is essential for preserving the timing impacts of an earlier instruction on a later one. The data and control dependency information of an in-order processor is contained in various pipeline registers, which has to be extracted for a snapshot. Later, in Section 5, we explain how a *shadow pipeline* is used to extract and preserve dependencies. For now, we can consider the *shadow pipeline* as a small set of registers containing the dependencies, as shown in Figure 3(a)(b).

Stall cycles in Decode stage and Memory stage: For in-order processors, only some stages will incur variable stalls. In our native processor, for example, the Decode stage (where all instructions are issued) and the Memory stage (where memory instructions are issued) are such stages. According to the theorem presented in [14], the entry/exit times of these stages need to be captured. However, we observe that capturing only stall cycles suffices to preserve worst-case timing due to strict in-order execution. In fact, the timing dependencies below imply that structural dependencies can be preserved in this manner:

$$IssueTime(inst_i) = EntryTime(inst_i) + StallCycles(inst_i) + 1 \quad (1)$$

$$EntryTime(inst_{i+1}) = IssueTime(inst_i) \quad (2)$$

where $inst_i$ and $inst_{i+1}$ are two instructions dispatched successively to a same pipeline stage (the Decode or Memory stage in our case). These equations imply that structural and data dependencies are preserved for in-order processors by recording only the stall times.

The exact number of stall times to be captured in snapshot is also microarchitecture-dependent. For now, we loosely refer to them as "time tags" as shown in Figure 3(c), and will explain them in detail in Section 5.

Since for this work we do not consider the issue of cache analysis, values in the register file can be safely treated as *Don't Cares*. Instead, as explained in Section 5, all memory instructions will be stalled for the duration of the worst-case memory latency to obtain the pessimistic result. In addition, most of the contents in pipeline registers are irrelevant to the dependencies and can also be ignored.

4.2.2 Merging Snapshots of Multiple Execution Contexts

Another key component of our approach is the ability to merge multiple snapshots into one single snapshot when multiple paths join. Merging snapshots can help reduce the number of paths to enumerate during WCET analysis and largely save the analysis time. The objective here is to preserve the worst-case timing when snapshots are merged [13, 14]. Pathological behavior as illustrated at the beginning of this section can be accounted by always preserving the impacts of data, control, and structural control dependency on timing. We next show how each type of these aforementioned dependencies is preserved during merging.

Merging pipeline states: The pipeline state in a snapshot, as described previously, is irrelevant to the worst case timing as long as timing information is correctly merged as explained below. As a result, we can use an arbitrary pipeline state in the merged snapshot. However, for convenience, we opt to use the pipeline state with longest worst execution cycle. This operation is illustrated as the *MUX* operation in Figure 3(c).

Merging timing information: The timing information is two-fold: the execution time up to the current snapshot and the stall cycles in stages with variable delay. The merge operation for timing information is to take the maximum values as a pair of corresponding timing tags in each snapshot. This operation is illustrated as the *MAX* operation in Figure 3(c).

Merging data/control dependencies: In theory, for the snapshot of a single execution context, the structural dependency will automatically preserve the data and control dependencies. However, when multiple snapshots are merged, the dependency information contained in the *shadow pipeline* registers of all snapshots has to be preserved. This can be viewed as a *UNION* operation as illustrated Figure 3(c). To preserve data dependencies, we instrument the *shadow pipeline* stages with extended registers to hold the additional information contained in a merged snapshot. Each register has an 8-bit register file address field and a valid bit (the "L" bit). Since global register 0 does not impose data dependencies, and the *shadow pipeline* registers can hold the data dependencies of one snapshot, we only need 30 extended registers associated with each *shadow pipeline* stage to account for the maximum number of registers (32 registers in a SPARC V8 register window).

The control dependencies are represented by the control bits in the *shadow pipeline* registers. It is straightforward to see that the *UNION* operation for these control bits can simply be reduced to a bit-wise *OR* of corresponding bits. The outcome can be directly put into the control bits of the merged snapshot, and no additional registers are needed.

While replaying a snapshot, the preserved dependencies contained is enforced by the interaction between the *shadow pipeline* and the original pipeline, which will be detailed in Section 5.

4.3 Summary of Snapshot

The format of snapshot in our design has to account for cases of both single execution contexts and multiple execution contexts. Figure 3(c) summarizes the main blocks of the snapshot and the operations needed to merge two snapshots. The theorem presented [14] can be used to prove that this operation can be carried out recursively to merge an ar-

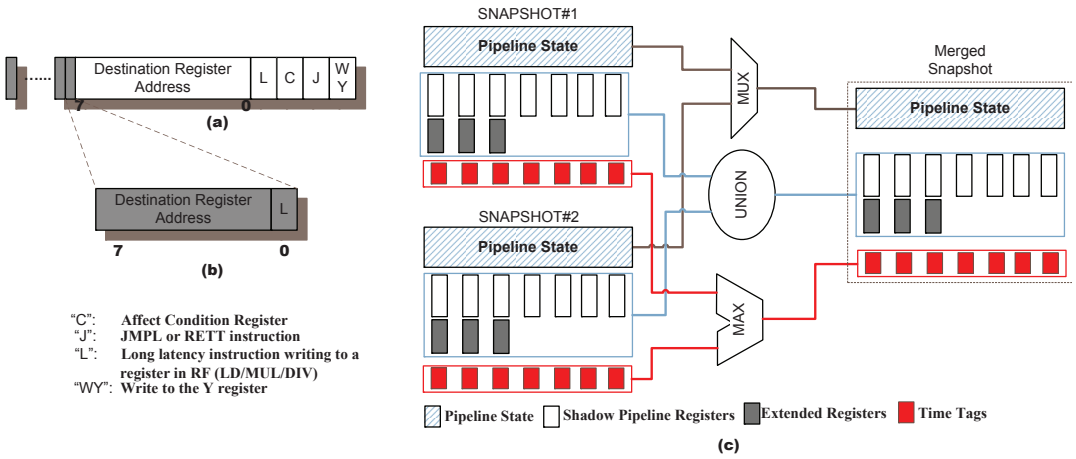

Figure 3: (a) Shadow pipeline register (b) Extended register whose format is actually a partial shadow pipeline register (c) Merge snapshots

bitrary number of snapshots. The *shadow pipeline* has the same depth as the unmodified native pipeline, which has seven pipeline stages in our case. However, only the first three stages of the *shadow pipeline* are associated with extended field, because in our current microarchitecture only one bubble may be inserted by a long latency instruction (Section 5. The timing tags include both the execution time and the stall time. The next section will provide more detailed information about the microarchitecture and how the snapshot is mapped to the components in the CheckerCore.

5. HARDWARE IMPLEMENTATION

In this section, we discusses the specifics of the microarchitecture of CheckerCore. The overview of the CheckerCore architecture is shown in Figure 4, in which a native processor (the black dotted box) is enhanced by other CheckerCore components (other components shown in the figure). The major CheckerCore components include a *shadow pipeline*, a *synchronization logic* that coordinates the instruction flow in both pipelines, a *timer block*, a centralized *CheckerCore Controller*, and a *snapshot buffer*. The synchronization logic partially overlaps with the timer block, to drive both pipelines in a way that preserves the worst-case timing during replaying a snapshot. Most interconnects in the CheckerCore are neglected in this figure for clarity.

5.1 LEON3 Processor

We enhance the LEON3 processor with the proposed CheckerCore infrastructure. LEON3 is an open source SPARC V8 soft-core developed by Gaisler Research [8]. It features a seven-stage in-order pipeline (the black dotted box in Figure 4). As discussed previously, control and data dependencies are resolved in the Decode stage and variable stalls may be introduced; memory instructions are issued to the data memory at the Memory stage where they could potentially stall. In practice, the instruction and data memories are shared and they both reside in the external DDR2 SDRAM.

Dependencies affect timing in LEON3 as follows. Long latency instructions (load/mul/div) will typically introduce *one bubble* in the pipeline. Hence, only a long latency instruction in the Register stage may potentially cause following instructions to stall due to data dependencies. Instructions that affect branch conditions may finish as late as at the Memory stage that could potentially introduce three bubbles if a conditional branch follows. The Y register (used by mul/div instructions) is also written at the end of the Memory stage and any instruction that reads this register may also be stalled for three cycles at the Decode stage.

One interesting feature of LEON3 is that, some instructions will be translated into two or more short instructions at the Decode stage. *E.g.*, a load double-word instruction will be decomposed into two load single-word instructions. This feature affects how we calculate the stall cycles. The following sections (5.2, 5.3, 5.4, 5.5, 5.6, and 5.7) provide details on the enhancements/additions made to the LEON3 to adapt it for CheckerCore functionality.

5.2 Shadow Pipeline

To extract data and control dependencies we instrument the original LEON3 with a so-called shadow pipeline (the blue dotted box in Figure 4). As shown, the only logic in the shadow pipeline is a simplified decoder that extracts the destination register address and several other control bits. As a result, the format of the shadow pipeline registers is also very simple (Figure 3(a)). In effect, the shadow pipeline duplicates the data and control dependency information without disturbing the native pipeline. When a snapshot is taken, the shadow pipeline can be frozen and the information can be read out by the CheckerCore controller.

The role of the shadow pipeline is actually two-fold. During normal execution and the process of taking a snapshot it is driven by the native pipeline and extracts dependencies. While replaying a snapshot the shadow pipeline is loaded with dependency information and passes this information to the dependency resolution logic of the native pipeline. Hence, the shadow pipeline is used for *both capturing and preserving* the dependencies. In either case, the shadow pipeline is synchronized with the native pipeline by the synchronization logic. The shadow pipeline is also associated with extended registers whose format is shown in Figure 3(b). The extended registers are used purely for preserving the additional data dependencies in a merged

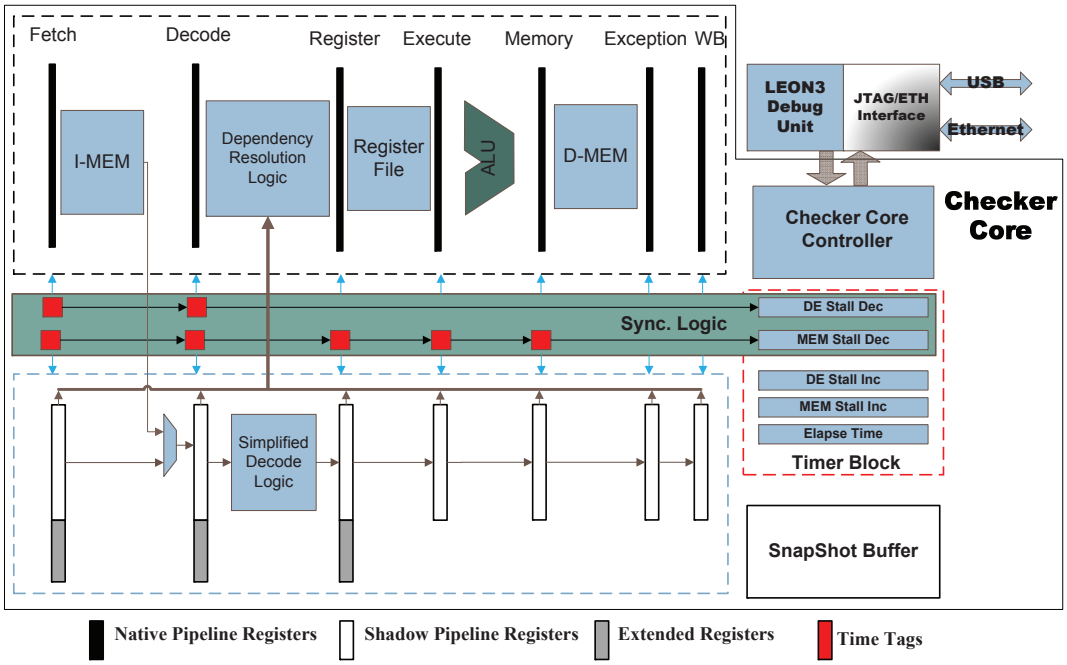

Figure 4: Overview of micro-architecture of LEON3 processor enhanced with Checker Mode

snapshot after taking the *UNION* operation, as explained in the previous section. The "L" bit associated with each destination register address in both the extended as well as the shadow pipeline registers simply indicates that there is one long latency instruction with this destination register at a given stage. This information is used by the native pipeline to insert bubbles. Only the first three stages of the shadow pipeline have extended registers since only one bubble may be inserted into the pipeline due to data dependency, as discussed earlier in this section . During replay, the extended registers are reloaded by the CheckerCore controller and run in the same way as the shadow pipeline, *i.e.*, synchronously with the native pipeline.

5.3 Synchronization Logic

In both the normal execution mode and the CheckerMode, the main function of the synchronization logic is to ensure that both pipelines run at the same pace. However, the role of this function is slightly different during normal execution and while replaying a snapshot. During normal execution or the process of capturing a snapshot (in CheckerMode) the synchronization logic is driven by the native pipeline and controls the advance of the shadow pipeline. However, while replaying a snapshot, the synchronization logic controls both the native pipeline and the shadow pipeline. The advancement of both pipelines not only depends on the natural execution of the native pipeline but also depends on the worst-case timing information contained in the snapshot. This is achieved by *time tag registers* and the two decrementing timers (DE Stall Dec and MEM Stall Dec in Figure 4). The time tag registers are organized as two shift register chains each corresponding to a decrementing timer. The two decrementing timers will assert corresponding underflow signals if they have counted to zero. While replaying a snapshot the synchronization logic will stall the Decode or Memory stage until the DE Stall Dec or the MEM Stall Dec timer underflows. When either decrementing timing underflows, the synchronization logic will reload it with the value in the last time tag register of the corresponding time tag chain and also shift forward the time tag chain. The time tag registers are essentially loaded with the stall cycles from a snapshot before replaying it.

Moreover, as we will see in the next subsection, the MEM Stall Dec can be loaded with a pre-programed memory latency for each memory instruction during normal execution. This function is provided for the cases that cache analysis is not carried out (as in our case). If this function is enabled the synchronization logic will also stall the memory stage for a memory instruction during normal execution until the timer underflows.

5.4 Timer Block

The timer block is essential for capturing timing information and preserving the worst case timing when a snapshot is taken and when a snapshot is replayed, respectively. The timer block contains five timers. The two decrementing timers DE Stall Dec and MEM Stall Dec are used to preserve the worst-case timing during snapshot replay and have been explained previously. The three incrementing timers, DE Stall Inc, MEM Stall Inc and Elapse Time are used for capturing the timing information of a snapshot. Specifically, when a snapshot is being captured, the DE Stall Inc and MEM Stall Inc will count the stall cycles of the instructions passing through the Decode and Memory stages. The result is read out synchronously by the CheckerCore Controller when an instruction leaves the corresponding stage, which will then reset the timer to count the stall cycles of the next instruction. The Elapsed Timer is used to count the execution cycles between two snapshots. Finally, the two decrementing timers will load different values during differ-

ent scenarios. While replaying a snapshot, both timers will load the value from the time tag chain. However, during normal execution, the DE Stall Dec will always reload 0 (thus it always underflows) while the MEM Stall Dec may load 0 or the pre-programmed memory latency depending on whether cache analysis is available and if a memory instruction is issued. Except for the Elapsed Time timer (32-bit), all other timers are 8-bit in length.

The calculation of stall cycles deserves some explanation. As mentioned at the beginning of this section some instructions will be translated into a number of micro-instructions at the Decode stage. Our CheckerCore does not time the stall cycle of each micro-instruction but treats all the micro-instruction generated by a single instruction as an single entity. The stall cycles of that instruction is calculated by subtracting the entry time of the first micro-instruction into a pipeline stage from the issue time of the last micro-instruction from that particular pipeline stage. During snapshot replay, if the last micro-instruction leaves before the corresponding decrementing timer completes counting down then the next instruction will be prevented from entering the Decode or Memory stage until the timer reaches zero. Hence, the worst-case stall times of instructions that generate micro-instructions can be preserved during replay.

5.5 Snapshot Buffer

The snapshot cache serves as an on-chip cache to hold recently captured snapshots or those that must be restored shortly. This is to improve the performance of WCET analysis. This buffer is actually managed remotely by the CheckerCore driver that we describe in further detail in Section 7.

5.6 CheckerCore Controller

The CheckerCore controller drives the execution of the microarchitectural additions. It implements communication protocols with the CheckerCore Driver (Section 7) and coordinates the actions of all components in CheckerCore. It is also capable of steering the pipeline to run on different paths controlled by the front-end.

5.7 LEON3 Debug Unit with JTAG/Ethernet Interface

We enhance the LEON3 debug unit with integrated JTAG/Ethernet Interface. The original LEON3 debug unit communicates with the JTAG/Ethernet controller via the same bus that the processor uses to access the main memory and other devices. To eliminate contention between diagnostic messages and regular execution, we remove the debug unit and the JTAG/Ethernet controllers from the bus and integrate them as a single entity.

5.8 Summary of CheckerCore

With the above enhancements, CheckerCore is capable of *(a)* capturing snapshots, *(b)* replaying snapshots and *(c)* steering the execution while still preserving worst-case behavior in all cases. During snapshot capture both the native and the shadow pipeline are drained. This allows the shadow pipeline to obtain dependency information that is then read out to the snapshot buffer. The decode and memory stall cycles of each instruction are also captured in the snapshot buffer as part of the drain process. Before restoring a snapshot a previous snapshot is used to put the processor in a consistent state. The processor then continues execution until the snapshot that is to be restored is reached. This warms up the processor state for restoring. The shadow pipeline is then loaded with dependency information and the time tag chains are reloaded with the stall cycles of the snapshot. After that, the processor continues execution while the synchronization logic guarantees the preservation of the worst-case behavior of the snapshot. Steering the execution along required paths is made possible by the CheckerCore Controller that provides target PCs and branch conditions to override the behavior of the native pipeline for jumps and conditional branches. The target PCs and branch conditions are provided by the front-end.

The size of a snapshot is determined by the characteristics of a processor. In our case, a snapshot contains *(a)* three extended registers, *(b)* five shadow pipeline registers, *(c)* two time tags for the Decode stage stall (of which at most two instructions will pass through the Decode stage during snapshot capture/replay), *(d)* five time tags for Memory stage stall (of which at most five instructions will pass through the Memory stage during snapshot capture/replay), *(e)* the pipeline state, and *(f)* the elapsed time. The total size of the snapshot, including padding overhead, amounts to 192 bytes for our current CheckerCore implementation. As a result, at most *five* snapshots can be read from and written to the CheckerCore in a single Ethernet transfer.

6. FRONT-END FUNCTION

The front-end, also referred to as the Timing Analyzer (TA), constructs CFGs of the executable, issues commands to the CheckerCore for timing sections of the program and derives the final WCET for the program. This section gives an overview of the process of determining the WCET for a program section and the commands exchanged between the TA and the CheckerCore during the process.

6.1 Overview of Bounding WCETs

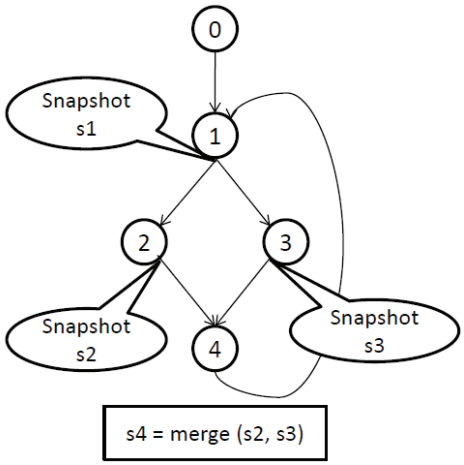

Figure 5: If-else loop.

Prior work in literature described a hybrid method to measure the WCET of the program [13, 14]. We briefly summarize it here, to facilitate understanding, since we implement a similar scheme in CheckerCore. Consider a loop such as

the one shown in Figure 5 consisting of four basic blocks. The WCET of the loops is determined as follows. We start execution from basic block 0 and capture a snapshot (s1) at the end of basic block 1. Snapshot capture drains the pipeline, so before we can time the path 1–2, we need to *restore* s1. Restoring a snapshot has two components. We begin execution from a previous point, say basic block 0, so that the pipeline is full when execution reaches basic block 1. The 'previous point' is usually the previous snapshot, if one exists. We then use the timestamps of s1 to enforce pathological timing behavior. This is also referred to as *replay* of snapshot s1. Now, CheckerCore is steered through one of the paths, say 1–2, and a snapshot (s2) is taken at the end of basic block 2. In order to time the other path, s1 is restored, execution is steered through the other path 1–3 and a snapshot(s3) is taken at the end of basic block 3. Now, snapshots s2 and s3 are merged to get s4. Finally, to time one iteration of the loop, s1 is restored. The execution is steered either through path 1–2 or path 1–3 and snapshot s4 is replayed. The merge algorithm [14] ensures that the pathological effects of both paths are retained post merge. Hence, it is sufficient to re-execute only one of the paths (essentially the path with the *longer* execution time) for timing purposes. The execution time of the path is recorded and relayed to the TA. For timing subsequent iterations of the loop new snapshots s1, s2, s3 are taken in each iteration after replaying snapshots (s1 and s4 in this case) taken at the previous iteration. This procedure is continued until the run time of the loop reaches a fixed point. After the fixed point is reached, the run time is extrapolated for the remaining loop iterations [1].

6.2 Interaction Between the Timing Analyzer and CheckerCore Controller

Recall that the timing analyzer creates the CFGs and instructs CheckerCore to measure the run time of various paths. The CheckerCore controller manages the CheckerCore (enhanced FPGA soft core). We now present an API command language that facilitates the interaction between the two components, the front-end TA and the CheckerCore Controller that ultimately provides access to the back-end.

According to their different impacts on the back-end, commands can be classified into two groups:

I. Setup Commands: These commands specify the execution environment. No instructions of the target program are executed as a result of these commands.

II. Execution Commands: These commands trigger execution of instructions from the current state of the pipeline. The state itself is only influenced by setup commands. Results from the execution are returned to the TA.

6.2.1 Setup Commands

These commands set up the processor state before executing instructions along a path:

1. put snapshot <Snapshot ID>: This command specifies a snapshot from which execution needs to be restarted. A PC, a register window (and other architecture specific information) included in the snapshot is used to restart execution from a particular point in the program. The snapshot is identified by a unique snapshot ID. If the snapshot is not cached within the CheckerCore cache, the CheckerCore controller obtains the snapshot from the TA which has a larger buffer that stores all previous snapshots.

2. put timing <Snapshot ID>: This command replays a snapshot identified by <Snapshot ID>. If the snapshot is not cached within the CheckerCore cache, it is obtained from the TA.

3. put PC <Branch PC><Next PC><T/NT>: This command overrides the outcome of control flow instructions and thereby forces branches to evaluate such that control flow is directed along the desired path. It has three arguments: the branch PC, the PC of the target instruction and the branch outcome — taken/not taken.

These commands can be issued once they are interpreted in a sequential manner. Multiple "put PC" commands may have to be issued to steer the CheckerCore through a particular path in the program. Also, multiple "put timing" commands can be issued to replay a sequence of snapshots.

6.2.2 Execution Commands

These commands result in execution of instructions through the pipeline:

a. get snapshot <Snapshot PC>: The command captures snapshots. It initiates execution from the current "state" of CheckerCore. The "state" itself is affected by the commands above. When <Snapshot PC>is fetched from the instruction stream a snapshot is captured and returned to the TA. Note that a PC specification does not uniquely identify a single point during program execution. PCs repeat iteratively within loops or when functions are called at different call sites (or within loops). However, snapshots have to be uniquely identified with points during program execution. This is realized by ensuring that all "put PC" commands are completed before a unique point in execution is reached and a snapshot is captured. For example, in order to take a snapshot at the beginning of the third iteration of a loop the "put PC" command is issued twice before the "get snapshot" command. The two "put PC" directives steer execution into the third iteration of the loop before a snapshot is captured.

b. get timing <PC1><PC2>: This command captures execution times for program paths. Execution of instructions is initiated from the current "state". Subsequently, the time taken to execute all instructions between the specified PCs (and hence for that particular path) is returned to the TA.

7. EXPERIMENTS AND RESULTS

7.1 Experimental Setup

Our baseline architecture is LEON3, a SPARC V8 implementation from Gaisler Research [8] The baseline architecture is configured to support the full SPARC V8 specification, except for co-processor and floating-point instructions. Both the I-cache and D-cache are also turned off and, as explained previously, we stall all memory instructions for a pre-programmed number of cycles. Cache and memory analysis is orthogonal to our framework and can be added as part of future work.

The enhancements proposed in Section 5 are applied to the baseline architecture and implemented on a Xilinx

[1]The fixed-point approach for loop analysis is elaborated in a different paper that is currently under submission and is not presented here due to lack of space. Any such technique(s) can easily be integrated into CheckerCore to obtain more comprehensive WCET results.

ML505 FPGA evaluation board, which hosts a Xilinx Virtex 5 FPGA chip (XC5VLX110T). We connect the board to a computer using the 10/100 Mbps Ethernet physical interface and use the Ethernet debug interface shown in Figure 4 for communication between the front-end and the back-end.

The final hybrid software/hardware platform is assembled by a so-called "CheckerCore Driver", analogous to a device driver. This driver implements the command interface from Section 6. When analyzing the WCET, the driver accepts commands from the timing analyzer (TA) that drives the execution in CheckerMode in the enhanced processor. It also collects snapshots resulting from CheckerMode execution and relays them to the TA for further processing. The interaction between TA and CheckerCore generates the final timing results presented next.

7.2 WCET Estimation Result

Using the system mentioned in Section 7.1 we derive the WCET estimation of 6 benchmarks (Table 1): *simple* and *toy* are synthetic benchmarks that we constructed; *cnt*, *bs*, *factorial*, and *fibonacci* are from C-Lab benchmark suite. Benchmark *simple* has only an if-then-else block. It is the simplest benchmark that exercises our merge algorithm. Benchmark *toy* has a simple if-then-else block and a loop of 10 iterations within the else block. The CFG of *toy* is a bit more complex than *simple* but requires fewer snapshot merges than *cnt*. Benchmark *cnt* finds the sum of positive and negative numbers in an array of size 10. Benchmark *bs* is a binary-search in an array of size 15 elements. Benchmark *factorial* finds 5! using an iterative loop. Benchmark *fibonacci* finds the fibonacci number of 30 using an iterative loop. To execute each benchmark, we first fast-forward to the start of the "main" function, and then start the process of capturing the WCET until the end of that function. Fast-forwarding is enabled by CheckerCore. A *new command to initiate fast-forwarding* is added to the driver besides those mentioned in Section 6.

Table 1 lists the results (worst execution cycles) for each benchmark. In our experiments we varied the programmed memory stall cycles, and for each value we obtain a set of results to study the impact of memory latency. Rows 2-5 of the table show four sets of results corresponding to the memory stall cycles of 0, 8, 16, and 32, respectively. This is carried out for all six benchmarks. We notice that for all the benchmarks the WCET increases, as expected, with the increased memory latency.

Hence, we see that CheckerCore is able to calculate the WCETs for embedded benchmarks without requiring the use/development of a timing accurate simulator. This also shows that the proposed architectural enhancements are realistic and impose low overhead. The exact overheads for CheckerCore are discussed in the next subsection.

Table 1: Worst Case Execution Cycles of Benchmarks

Stall Cycle	simple	cnt	bs	factorial	fibonacci	toy
0	173	6610	2029	467	623	2139
8	188	7258	2134	563	886	2434
16	224	8500	2682	722	936	3022
32	309	11484	3874	1066	1336	4189

Table 2: Comparison of Implementation Logic

	LUT Flip Flop	Slice LUT	Block RAM
Available	69120	69120	148
LEON3	12399	11450	10
CheckerCore	14260	12855	11
Overhead	15%	12%	10%

7.3 CheckerCore Overhead

Table 2 shows the comparison of resource utilizations of the baseline LEON3 and the CheckerCore. CheckerCore uses a significantly larger number of LUT (Lookup Table) Flip-Flops However, given the abundant flip-flop resources on the Virtex 5 FPGA this overhead is negligible. CheckerCore also uses one more Block RAM (BRAM) to implement the snapshot buffer.

8. RELATED WORK

CheckerMode [13, 14] is an approach to capture advanced hardware features transparently while providing tight WCET bounds. It introduced a hybrid timing analysis technique of OOO processors including Drain Retired Merging (DRM) through architectural simulation at the cycle level. Our work differs in that it simplifies the DRM approach for in-order processors. While this work still relied on the use of timing accurate simulators, we demonstrate the feasibility of the approach by integration into an FPGA soft core.

Bernat *et al.* [3] used probabilistic approaches to express execution bounds down to the granularity of basic blocks that could be composed to form larger program segments but suffered from considerable timing perturbation. CheckerCore, on the other hand, can be quite precise in its measurements. The VISA framework [1] suggested architectural enhancements to gauge progress of execution by sub-task partitioning and exploit intra-task slack with DVS techniques. The *virtual processor* in VISA, which both enables performance improvement and guarantees easy WCET analysis, is composed of a simple pipeline and a complex core. In this paper, 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. Instead of a VISA-like *virtual processor* around a complex core, we demonstrate that CheckerCore is a realistic feature building on modular extensions with much lower overheads. Hence, our method is more precise than Bernat's work while, in contrast to VISA, supporting hybrid timing analysis on the actual processor core as evidenced in our soft core enhancements.

Lundqvist et al. [10, 11] use symbolic execution, a tight integration of path/timing analysis and the concept of an *"unknown"* value to account for register values and addresses that cannot be statically determined to obtain accurate WCET estimates. However, their work focused on static timing analysis over the entire program within an architectural simulator. CheckerCore obviates the need for such accurate timing simulators. Whitham [18] presents a combination of hardware and software techniques to capture WCETs accurately for complex processors. Instruction scheduling is carried out by the compiler and relies on custom microcode executing in the processor. However, this approach

requires a redesign the native pipeline and the entire toolchain. In contrast, CheckerCore reuses most of the existing infrastructure, such as the unmodified compiler and the native pipeline, and only adds modular components that incur low overheads (Table 2).Some early work has suggested probabilistic WCET analysis [3, 5, 9, 16] to consider WCET variations due to implicit factors (such as data dependencies and architectural features) without detailed modeling of the hardware. However, these approaches cannot guarantee that the calculated timing values have not been underestimated, a situation that can result in dangerous fallouts such as missed deadlines.

9. CONCLUSION

Accurate WCET estimation for embedded processors is critical for hard real-time embedded systems. In this paper, we propose a hybrid approach that combines both static and dynamic approaches. Our WCET analysis system is comprised of a traditional front-end that deconstructs and analyzes program execution paths and a back-end that is a native target processor enhanced by a novel "CheckerCore" synthesized onto an FPGA.

The most important difference between our approach and prior work is the introduction of CheckerCore, which uses the real hardware as the timing engine. Beside the advantage of hiding microarchitectural details, it provides three additional benefits: *(i)* CheckerCore eliminates the need for a separate simulator and associated development and verification costs. Although the design of CheckerCore is subject to microarchitectural changes, software simulators share the same problem. Also, the latter requires significant efforts for verifying consistency that is absent in the former. *(ii)* CheckerCore largely improves the speed of analysis. Although software simulators can trade off accuracy with performance, for cycle-accurate simulators the speed is still about several $100K$ instructions per second [2, 12]. This problem is further aggravated for full-system simulators. In contrast, CheckerCore runs at the same speed as the native processor that is at least faster by an order of magnitude (several million instructions per second). *(iii)* CheckerCore is unique in that it can analyze processors that are subject to variations of manufacturing technology but can still perform analysis on the actual silicon during execution, which is unprecedented. To illustrate the feasibility and benefits of CheckerCore, we designed and implemented a hardware/software hybrid platform for WCET analysis based on this architecture.A prototype of the CheckerCore idea is shown to be successfully implemented with a SPARC V8 processor softcore synthesized on an FPGA platform. In future work, we intend to focus on the automatic generation of CheckerCores for a given microarchitecture. This should help in reducing the time/cost of such enhancements.

10. REFERENCES

[1] 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.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb 2002.

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

[4] V. Braberman, M. Felder, and M. Marre. Testing timing behavior of real-time software. 1997.

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

[6] R. W. et al. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, Apr. 2008.

[7] S. M. et. al. Parascale: Expoliting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *IEEE Real-Time Systems Symposium*, pages 233–242, Dec. 2005.

[8] A. Gaisler. Leon3 product sheet. `http://www.gaisler.com/doc/leon3_product_sheet.pdf`, September 2008.

[9] 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.

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

[11] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2/3):183–208, Nov. 1999.

[12] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb 2002.

[13] S. Mohan and F. Mueller. Hybrid timing analysis of modern processor pipelines via hardware/software interactions. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 285–294, 2008.

[14] S. Mohan and F. Mueller. Merging state and preserving timing anomalies in pipelines of high-end processors. In *IEEE Real-Time Systems Symposium*, Dec. 2008.

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

[16] 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.

[17] 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.

[18] J. Whitham. *Real-time Processor Architectures for Worst Case Execution Time Reduction*. PhD thesis, University of York, May 2008.