CheckerMode : A hybrid scheme for timing analysis of modern processor pipelines involving hardware/software interactions

Sibin Mohan and Frank Mueller
Dept. of Computer Science, Center for Embedded Systems Research,
North Carolina State University, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu

Abstract

Real-time systems often require determinism to ensure that task deadlines are met. Schedulability analysis provides a firm basis to ensure that tasks deadlines are met, and for this, knowledge of bounds on worst-case execution times (WCET) of tasks is a critical piece of information. Static timing analysis derives these bounds on WCETs. A limiting factor for real-time systems design is the class of processors that may be used. Contemporary processors with their advanced architectural features, such as out-of-order execution, branch prediction, speculation, and prefetching, cannot be statically analyzed to obtain WCETs for tasks because these features introduce non-determinism to task execution, which can only be resolved at run-time. We introduce a new paradigm which proposes minor enhancements to modern processor architectures, which, on interaction with software modules, is able to obtain tight, accurate timing analysis results for modern processors. To the best of our knowledge, this method of hardware/software interactions to calculate WCET results for out-of-order processors is the first of its kind.

1. Introduction

Embedded systems are increasingly deployed in safety-critical applications and environments, such as avionics, power plants, automobiles, *etc.* The software used, in general, must be validated . This traditionally amounts to checking the correctness of the input/output relationship. Many such 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 ("deadlines") on the computation to ensure that necessary results are provided on time. The worst-case execution time (WCET) of each task is one critical piece of information required by real-time systems designers to verify that tasks meet their deadlines.

Static timing analysis [3–7, 9] provides bounds on the WCET. The *tighter* that these bounds are relative to the actual worst-case times, the better the value of the analysis. Of course, any tight bound has to be *safe* in that it must *never* underestimate the true WCET; it may only match or exceed it.

A serious handicap in performing static timing analysis is the complexity of modern processors and their functional units. Out of order (OOO) processing [8], branch prediction [10] *etc.* introduce non-determinism to task execution that cannot be resolved at compile time. Hence, designers of real-time systems are often forced to use older, less complicated and inherently less powerful processors. In this paper, we attempt to bridge this gap by the use of the *CheckerMode* infrastructure.

We propose minor enhancements to the micro-architecture of future processors that will aid the processes of obtaining tight WCET bounds. A "checker mode" is added to processors that will, on demand, capture varying details of the processor state (called "snapshots"). This information is communicated to a software module that stores the snapshots and also drives the execution of the processors along statically determined paths to capture accurate timing information for each of them. The snapshots are used to track back along the various execution paths and to restart along a different path if necessary. The execution times obtained for each of the paths is analyzed and combined by the software driver to calculate an accurate WCET for the entire module/program. Decisions on where to obtain snapshots, the details required for a snapshot, *etc.*, are made by the software "driver".

The CheckerMode concept (implemented on an enhanced SimpleScalar processor simulator [2]), widens 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 a real-time system. We believe that this also changes the landscape for timing analysis as more accurate results can be obtained on modern pipelines without loss of functionality. To the best of our knowledge, this method is the first of its kind in using a hardware/software co-design technique to obtain accurate WCETs for modern, out-of-order processors.

This paper is organized as follows. Section 2 discusses the CheckerMode idea, while section 3 talks about the experimental setup and preliminary results. Section 4 summarizes the work.

2. CheckerMode

We use hardware/software interactions to perform WCET analysis of contemporary processors. We propose enhancements to embedded processors that, in addition to executing software normally (in "deployment" mode), are capable of executing in a novel *CheckerMode* that supports timing analysis. 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 depending on input data. In CheckerMode a processor 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, named "snapshot" (caches, branch history tables etc.), is restored to correctly simulate the effect of alternations in isolation from one another. The timing information as well

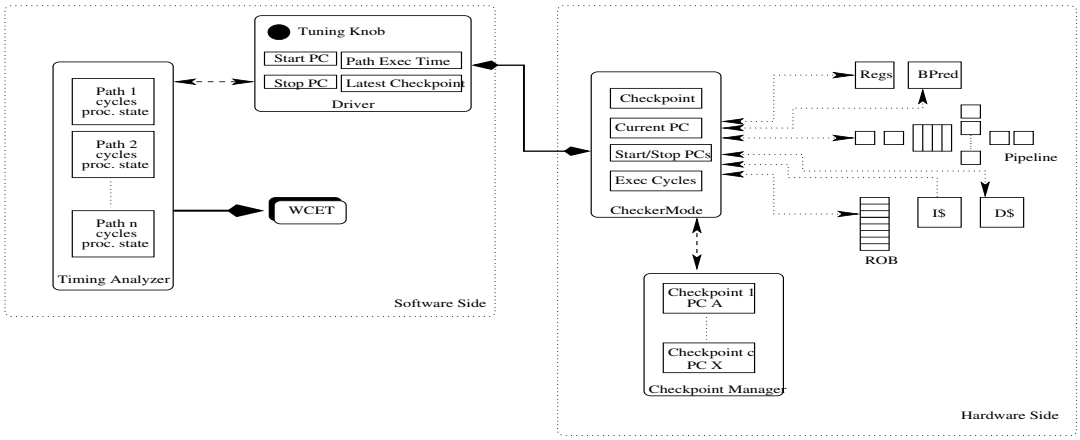


Figure 1. CheckerMode Design for High-Confidence WCET Analysis

as the "state" of the processor are combined when alternate paths join. The combination ("merge") 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. These low-level WCET results are propagated inter-procedurally in a bottom-up fashion until the WCET for an entire task has been computed.

We will represent input-dependent register values as "NaN" (not-a-number) values. Operations on unknown values are straightforward: if any input is unknown then the output is also unknown, even for 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 will alter the semantics of execution (for instructions that depend on input-dependent or memory-loaded operations) 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, for *e.g.*, multiplication with 0 will always result in 0).

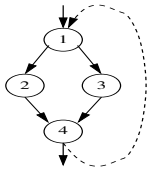
2.1. Overview of the framework

The hardware-supported CheckerMode is complemented by software analysis to govern execution (Figure 1). The **analysis controller (or driver)** steers 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 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).

Processor enhancements: The embedded hardware is also enhanced to support 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 of the path. The novel **CheckerMode** unit of the processor supports the following functions (right-hand side of Figure 1): *(a)* Capture *snapshots* of the processor state and communicate them to the software controller. Snapshots capture the current state of the pipeline, functional units, caches, ROB, etc. *(b)* Reset the processor to a previously saved state. The state of the pipeline, caches, functional units, *etc.*, is overwritten with information from the stored checkpoint. *(c)* Start and stop execution between arbitrarily provided 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 CheckerMode tracks the execution time for a given path (delineated by start and stop PCs) and is controlled by the driver on the software side.

Software controller: The left-hand side of Figure 1 illustrates the various components that make up the software side of the design. It consists of the following components: *(a)* **Timing Analyzer (TA):** breaks down the task code into a control-flow graph (CFG) and then extracts path information from it. It 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.*(b)* **Checkpoint Manager (CM):** maintains various snapshots that have been captured as well as the PCs at which they were obtained. CM abstractions can be integrated into the processor as depicted in Fig. 1, or, alternatively, into the driver within the software controller.*(c)* **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 a processor must be reset to a given snapshot.

The input to the TA is the executable of a task, which is then converted to internal representations. Start and stop PCs provided by the TA encapsulate a single path. The TA, the driver, and the CM interact to decide which checkpoint 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 for various program segments (functions/scopes). It "combines" the information from various paths (execution time/pipeline



(a) CFG

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

(b)Measured Cycles for Aggregate Technique

Figure 2. Control Flow Graph and Measured Cycles for Aggregate Technique

state/*etc.*) for this purpose.

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 checkpoints. The driver represents the interface between the hardware and software components and provides reconfigurability in terms of the amount of information to capture for the pipeline state and the state of associated functional units. We propose to provide a **virtual "knob"** that will allow real-time systems designers to tune the analysis, thereby trading off accuracy with overhead. We intend to explore the full design space of tuning options to assess which processor state information is more vital for WCET accuracy (and analysis performance) than some others. The more information is checkpointed, the tighter and more accurate WCET values will be. Conversely, less information will lead to a looser and more conservative WCET bound. Of course, greater demands on the amount of information being captured will lead to a slower WCET analysis whereas less information speeds up the analysis.

Reducing analysis overheads: We can reduce the complexity of determining WCETs 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 [1], we can steer loop executions such that paths of a loop body are repeatedly executed until a stable value is reached. The controller records the decaying execution times for each iteration up to the fixpoint using the hardware CheckerMode. When reaching the fixpoint, the WCET of the remainder of 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.

3. Experimental setup and Results

We have prototyped some of the key components of our design in the SimpleScalar processor simulator [2]. This cycle-accurate simulator can be configured for the various processor and branch prediction schemes mentioned in the previous section. Current enhancements include path-level timing capabilities and snapshot/restore of selected state information within the processor.

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

Notice that instructions are retired in order, even for OOO. Execution time for paths is measured using four different techniques, extending a basic block (BB) to paths (sequences of consecutive BBs): *(a) Short* measures the execution time for a singular BB, starting from the time that **any** instruction in the BB moves into the *execute* stage of the pipeline and finishing when the last of instruction of the BB exits from the *retire* stage; *(b) 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; *(c) 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; *(d) Path-Aggregate* captures the time for concatenated paths using the aggregate technique so that timing starts at the first BB and ends with the last BB of path.

The results obtained for the "short" and "path-short" techniques (numerical details omitted due to space) 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 to other instructions from prior BBs in the path. Even timing multiple BBs of a path in sequence does not alleviate this problem. In contrast, the "aggregate" technique (Figure 2(b)) reflects the time from instruction fetch (instead of execute), which addresses the above problem of early execution by some instructions. It shows a strict ordering of $SimIO \geq SupIO \geq OOO$, 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 excluding any pipeline overlap with prior BBs. Hence, these delta values can be used to assess the amount of cycles attributed to specific BBs. They also adhere to the same strict ordering. In general, such timings are only valid in the same execution context / path, *i.e.*, different BB sequences of one path may influence a subsequent BB in the control flow.

Our objective is to leverage path timings under the "path-aggregate" technique as a refinement to the "aggregate" technique discussed so far. Consider the construct depicted in

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 1. Program-Aggregate Cycles (3 Iterations)

Figure 2(a) embedded within a loop (dashed vertex) such that consecutive executions of paths can be assessed. *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 1 depicts the results for 3 iterations of this loop around the left (L) or 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 and indicates constant δ values for all processor models regardless of the paths executed. (D-caches are disabled here.) More significantly, early results within our experimentation environment indicate that 2-4 iterations generally suffice to reach a fix point. After that point, concatenation of another iteration results in a constant increase in cycles for this path that 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 fix point.

4. Conclusion

We have outlined a "hybrid" mechanism for performing timing analysis that utilizes interactions between hardware and software. This "CheckerMode" concept provides the foundation to make contemporary processors predictable and analyzable. These higher-end microprocessors can safely be used in real-time systems. Current trends in microprocessor features indicate that our proposed hardware modifications are realistic [11]. Once fully implemented within the SimpleScalar simulator, the CheckerMode unit will have the ability to drive execution along given program paths and also capture and writeback processor state to/from snapshots. It will also be able to accurately gauge the execution time for a given program path. We believe this work will enhance the choices available to real-time systems designers. The CheckerMode concept will provide them with the ability to use current and future microprocessors in their systems and utilize a hybrid of static and dynamic timing techniques to validate WCETs.

References

[1] 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.

[2] 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.

[3] 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.

[4] S. Malik, M. Martonosi, and Y.-T. S. Li. Static timing analysis of embedded software. In *Proceedings of the 34th Conference on Design Automation (DAC-97)*, pages 147–152, NY, June 1997. ACM Press.

[5] 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.

[6] 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.

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

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

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

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

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