Non-Uniform Program Analysis & Repeatable Execution Constraints: Exploiting Out-of-Order Processors in Real-Time Systems

Aravindh Anantaraman
Dept. of ECE, North Carolina State University
Campus Box 7256
Raleigh, NC 27695-7256
(+1) (919) 513-2014
avananta@ece.ncsu.edu

Eric Rotenberg
Dept. of ECE, North Carolina State University
Campus Box 7256
Raleigh, NC 27695-7256
(+1) (919) 513-2822
ericro@ece.ncsu.edu

ABSTRACT

The objective of this paper is to enable easy, tight, and safe timing analysis of contemporary complex processors. We exploit the fact that out-of-order processors can be analyzed via simulation in the absence of variable control-flow. In our first technique, Non-Uniform Program Analysis (NUPA), program segments with a single flow of control are analyzed on a complex pipeline via simulation and segments with multiple flows of control are analyzed on a simple pipeline via conventional static analysis. A reconfigurable pipeline with dual complex/simple modes mirrors the hybrid analysis. Our second technique, Repeatable Execution Constraints for out-of-ORDER (RECORDER), defines constraints that guarantee a single input-independent execution time on an out-of-order pipeline for program segments with multiple flows of control. Thus, execution time can be derived via simulation with arbitrary inputs.

1. INTRODUCTION

Tasks' worst-case execution times (WCETs) are derived using some form of timing analysis. Present-day timing analysis tools can efficiently analyze the cycle-level timing of simple in-order processors, deriving tight WCETs in the context of these simple processors. Although tight, these WCETs are large because of the low performance of the underlying simple processor. Replacing the simple processor with a contemporary high-performance processor may yield smaller WCETs, allowing (1) additional tasks to be safely scheduled and/or (2) task rates to be increased. However, statically deriving tasks' WCETs on contemporary processors is extremely complicated and, in some cases, intractable. Since WCETs cannot be safely derived on contemporary processors, these processors are typically excluded from hard real-time systems.

In this paper, we examine why it is difficult to statically analyze the cycle-level timing of a contemporary out-of-order processor. We then propose two novel techniques that exploit an out-of-order processor to reduce tasks' WCETs with respect to WCETs on a simple in-order processor.

An out-of-order processor examines a "window" of dynamic instructions to create a high-performance out-of-order instruction schedule. As processor pipelines become wider (to fetch/execute more instructions per cycle) and deeper (to increase clock frequency), larger scheduling windows are needed to expose more instructions to the dynamic scheduler. For example, the Pentium 4 fetches 3 micro-ops per cycle and has more than 20 pipeline stages. To support such a wide and deep pipeline, the microarchitecture supports as many as 126 in-flight instructions in the scheduling window.

Statically deriving the WCET requires examining a corresponding scheduling window in software, and anticipating the worst-case schedule that would be formed dynamically by the processor. Although the processor schedules instructions at run-time, the schedule can be deduced easily, *a priori*, if there is no control-flow and no variable or unknown latencies. In the absence of control-flow, there is only one path of dynamic instructions, hence only one schedule.

Control-flow increases the number of possible paths through the program, which causes a corresponding increase in the number of possible execution schedules. Each additional branch doubles the number of paths and the number of possible schedules. Returning to the Pentium 4 example, assuming a branch every 8 instructions, we can have as many as 16 branches in the window. Worst-case timing analysis has to consider 2^{16} (65,536) possible execution schedules in this case. As each new processor generation supports successively larger windows, timing analysis becomes intractable.

Deriving the WCET for an entire program is an even harder proposition. Analyzing the program as a whole is not as simple as sub-dividing the program into discrete scheduling windows, deriving corresponding sub-WCETs, and then composing an overall WCET from sub-WCETs. A naïve concatenation of sub-WCETs is inconsistent with the fact that a hardware scheduling window continuously shifts through the dynamic instruction stream. The only way to truly capture the performance of an out-of-order, continuous-window processor is to enumerate every possible execution schedule through the entire program. This is impractical for programs with variable (i.e., input-dependent) control-flow.

In this paper, we propose two complementary techniques that greatly simplify timing analysis of contemporary superscalar processors, allowing us to exploit out-of-order execution and multiple-instruction-issue to reduce tasks' WCETs.

Our first technique is called Non-Uniform Program Analysis (NUPA). It adopts a non-uniform approach to timing analysis, where each region of a program is individually analyzed in the context of a pipeline model that is most suited for that region, in terms of efficient and tight analysis. NUPA exploits the fact that program segments with a single flow of control have a single execution schedule. Thus, simulation can be used to derive their WCETs on a complex pipeline. On the other hand, program segments with variable control-flow are statically analyzed in the context of a simple pipeline. We adapt a previously proposed reconfigurable processor [1] so that the hardware mirrors

NUPA's non-uniform analysis. At run-time, the processor pipeline is configured to operate in a *complex mode* (that resembles the complex pipeline) for program segments that were analyzed for the complex pipeline, and is configured to operate in a *simple mode* (that resembles the simple pipeline) for segments that were analyzed for the simple pipeline.

Note that NUPA only partially exploits the complex mode. It uses the complex mode only for program segments that can be simulated (i.e., no variable control-flow), reducing their WCETs, while WCETs of other segments are not reduced.

Our second technique directly attacks the problem of analyzing variable control-flow on an out-of-order pipeline. The key idea is to simplify timing analysis by eliminating variable control-flow and its side effects. Eliminating variable control-flow yields a single path through the program, permitting simulation-based timing analysis.

Eliminating control-flow for tractable timing analysis was first proposed by Puschner and Burns in the *Single-Path Architecture* [3], but their technique is not sufficient for an out-of-order pipeline. They used conventional *if-conversion* (predication). Predication converts input-dependent control-flow into input-dependent data-flow. Executing both paths of a branch yields multiple potential producer instructions of a value. Ambiguity in who produces the value is referred to as *input-dependent data-flow* in this paper. There may be a later consumer instruction, after the predicated block, that depends on this value. It is now ambiguous as to when the consumer executes because it depends on which producer is the true producer and when that producer executes. Thus, input-dependent data-flow can cause variations in execution time and has to be accounted for by timing analysis. Unfortunately, analyzing input-dependent data-flow is just as complex as analyzing input-dependent control-flow.

We propose a strict set of constraints, called Repeatable Execution Constraints for out-of-ORDER (RECORDER), which facilitates easy timing analysis of an out-of-order pipeline by ensuring a single execution time that is independent of the program's input. The first constraint stipulates that the number of dynamic instances of an instruction be constant across program runs, independent of the program's input. This can be implemented by predicated execution, i.e., executing both paths of a predicated branch and using the results of the correct-path instructions and discarding the results of the wrong-path instructions. This constraint is similar to the *single-path architecture.*

The second constraint requires that a dynamic instance of an instruction execute at the same instant of time across different program runs, again independent of the program's input. This constraint ensures that execution time is not affected by any input-dependent data-flow. This is achieved by issuing/executing a dependent instruction only after *all* potential producer instructions have executed.

Thus, RECORDER guarantees a single execution time that is independent of the program's input, and this execution time is both the actual execution time and the worst-case execution time. Using RECORDER, the execution time of a program can be *recorded* once using arbitrary (random) inputs and this does not change across program runs. The uniqueness of RECORDER lies in the fact that it allows instructions to execute out-of-order within a program, while guaranteeing that a specific dynamic instance of an instruction executes at the same time across different program runs, independent of the program's input.

Note that RECORDER is only a set of constraints and there may be multiple ways to implement these constraints. In this paper, we show that a previously proposed predication technique for out-of-order pipelines, called phi-predication [4], satisfies RECORDER constraints.

A task which is predicated indiscriminately may have a larger WCET on a complex pipeline than the WCET of the original non-predicated task on a simple pipeline, defeating the purpose of using a high-performance contemporary processor. It is well known that indiscriminate predication may degrade performance due to resource contention from wrong path instructions. To address this, we propose a combination of NUPA and RECORDER. The idea is to phi-predicate a program segment only if its WCET on the complex mode is less than the WCET of the non-predicated counterpart on the simple mode. The better policy is selected accordingly: simulation of phi-predicated version on complex mode (RECORDER) vs. conventional static timing analysis of non-predicated version on simple mode. Other factors may also be weighed in the policy decision, such as power consumption or feasibility of phi-predication.

2. RELATED WORK

Hybrid timing analysis approaches have been previously proposed where different segments of a program are analyzed using different timing analysis techniques. Symbolic hybrid timing analysis (SYMTA), proposed by Ernst and Ye [2], is one such approach. In SYMTA, a program is divided into single-feasible-paths (SFPs) and multiple-feasible-paths (MFPs). Simulation is used to estimate WCETs for SFPs and conventional analysis is used to estimate WCETs for MFPs. NUPA and SYMTA use a similar hybrid analysis approach. However, SYMTA only works with simple processors because it is constrained to a single pipeline model. On the other hand, NUPA matches hybrid analysis to a hybrid pipeline underneath, thus exploiting the performance of a complex mode for SFPs via simulation, while still enabling efficient static analysis of MFPs on a simple mode.

The *Single-Path Architecture* [3] proposed by Puschner and Burns is closely related to RECORDER. In the single-path architecture, variable control-flow is removed by *if-conversion.* The idea is to force a single execution path through the program and use simulation to derive the WCET. The single-path architecture implicitly assumes that instructions are executed in strict program order. This assumption does not hold in an out-of-order pipeline. Due to out-of-order execution, a value guarded by a predicated branch may be forwarded at different times depending on when the candidate producer instructions complete and when the predicate is computed, leading to variation in execution time. As a result, the single-path architecture cannot guarantee a single execution time (in spite of ensuring a single path through the program) on an out-of-order pipeline.

We demonstrate the problem with the single-path architecture on an out-of-order pipeline, using the same example provided in their paper, shown below.

```
1. r1 = expr1;
2. r2 = expr2;
3. test cond;
4. movt rr, r1;
5. movf rr, r2;
```

Assume that instruction 3 is executed first (i.e., *cond* is evaluated first), followed by instruction 2 (*r2* is computed second), and then instruction 1 executes (*r1* is computed last). Based on the outcome of *cond*, only one of the two *mov* instructions (inst. 4 or inst. 5) actually writes into register *rr*. (Note that the destination registers (*rr*) of inst. 4 and inst. 5 must be renamed to the same physical register to ensure that later consumers of *rr* get the correct value.) If *cond* is false, inst. 5 executes as soon as *r2* is computed. Later instructions that consume *rr* can now execute without having to wait for *r1* to be computed. Similarly, if *cond* is true, inst. 4 executes, writing the value of *r1* into *rr*. Later instructions that consume *rr* can now execute without having to wait for *r2*. Summing up, later dependent instructions may execute at different times based on the outcome of *cond* and when *r1* and *r2* are computed. There can be variations in execution time even with a single path, depending on the outcome of inst. 3 (i.e., *cond*).

Moreover, it is difficult or impossible to employ *if-conversion* in certain scenarios (for example, function calls in branch paths). It might be impossible to convert a whole program into a single path. Finally, whole-program if-conversion may also result in inflated WCETs and high power consumption. RECORDER also has the above issues. Nonetheless, NUPA permits us to opt out of the *complex* mode when phi-predication is difficult or not beneficial and allows us to revert to the simple mode instead.

In the Virtual Simple Architecture [1], tasks are speculatively attempted on the complex mode of a reconfigurable processor. However, WCETs are still derived in the context of the simple mode. The complex mode only creates dynamic slack. Tasks' WCETs are not reduced with respect to the simple mode. On the other hand, NUPA exploits the complex mode to reduce the WCET. Note that NUPA and VISA are complementary techniques. Program segments that are analyzed on the simple mode in NUPA can be speculatively attempted on the complex mode with the VISA gauging mechanisms turned on. Thus, we can exploit the benefits of both techniques – reduced WCET due to NUPA and dynamic slack due to VISA.

3. NON-UNIFORM PROGRAM ANALYSIS

Non-uniform program analysis (NUPA) is a flexible timing analysis strategy that allows different segments of a program to be analyzed on different pipelines. The key idea is to match each program segment to the pipeline that is most suited for that program segment, in terms of a tight WCET and easy analysis.

In this paper, we adapt the reconfigurable pipeline proposed in VISA [1] to match NUPA's timing analysis. The reconfigurable pipeline has two operating modes: (1) a high-performance *complex mode* that features out-of-order execution, multiple-issue, and dynamic branch prediction, and (2) a static-timing-analysis-oriented *simple mode* that features in-order execution, single-issue, and static branch prediction.

We divide a program into multiple smaller segments based on the nature of control-flow. Program segments with no branches or input-independent branches have a single path (i.e., a single-feasible path), and segments with input-dependent branches have multiple paths (i.e., multiple-feasible paths). Path clustering [2] can be used to classify program segments based on the nature of control-flow.

NUPA exploits the fact that simulation with arbitrary program inputs can be used to derive WCET on an out-of-order pipeline if there is no control-flow and no variable-latency instructions. Segments with single-feasible-paths are analyzed via simulation in the context of the complex mode of the reconfigurable processor. Other program segments with multiple-feasible-paths are analyzed via conventional static timing analysis in the context of the simple mode.

We illustrate NUPA using the example control-flow graph (CFG) in Figure 1. The CFG shows two types of control-flow. In the top portion of the CFG, there is a single control-flow path. In the lower portion of the CFG, there are multiple control-flow paths. On the right-hand side of the figure, we show NUPA. Since there is a single path through the top part of the CFG, it is analyzed in the context of the complex pipeline. The remaining part of the CFG is composed of multiple control-flow paths and it is analyzed in the context of the simple pipeline. Contrast this with conventional timing analysis (shown on the left-hand side of Figure 1), where the entire program is analyzed on a simple pipeline. NUPA can exploit the high-performance complex pipeline to reduce the WCETs of program segments with single control-flow paths, compared to corresponding WCETs derived by conventional analysis, which uses the simple pipeline uniformly.

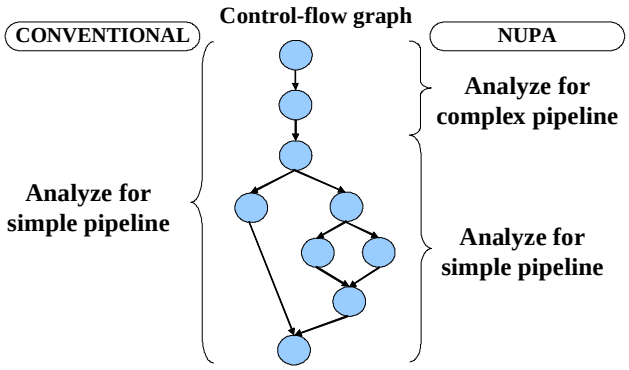

Figure 1. Conventional vs. non-uniform program analysis.

Variable-latency instructions, for example, memory accesses, multiply instructions, and floating-point divide instructions, can lead to different execution times in spite of a single-feasible path. For this initial work, we assume the worst-case latency for all variable-latency arithmetic instructions, independent of the program's input. For memory accesses, cache locking or software-managed scratch-pad memories can be used to guarantee a constant latency. Accounting for variable-latency instructions will be considered in future work.

Finally, to ensure a single execution time, the microarchitectural state at the start of a given program segment must be identical across program runs. This includes the state of the pipeline, the caches, and the branch predictor. We assume that the pipeline is drained at the beginning of a program segment. Also, the caches and branch predictor are flushed at the beginning of a program segment. The overheads of draining the pipeline and flushing the hardware structures are added to the WCET of the program segment.

The reconfigurable pipeline has to be configured to the complex mode at the beginning of program segments that are analyzed on the complex mode and switched back to the simple mode at the start of program segments that are analyzed on the simple mode. The overhead of switching has to be accounted for in the WCET.

The uniqueness of NUPA is that it can be applied at the task-set level too. For example, some tasks may have single feasible path from start to end. These tasks can be analyzed on the complex mode while other tasks with multiple feasible paths are analyzed on the simple mode. In this case, the pipeline is reconfigured only at task boundaries and not within tasks.

To demonstrate the benefits of NUPA, we use four tasks from the C-lab benchmark suite. In this paper, we only present results for task-set level NUPA. All 4 tasks have single-feasible paths and can be analyzed via simulation on the complex mode.

Task	WCET (ms)		% reduction in WCET w.r.t. simple
	simple	complex	
CNT	0.07	0.02	71%
FFT	0.36	0.06	83%
LMS	0.17	0.04	76%
MM	2.1	0.66	69%

Table 1. WCETs using NUPA.

Table 1 shows the WCETs of the 4 tasks on the simple mode (derived via static analysis) and the complex mode (derived via simulation with arbitrary inputs). The last column shows % reduction in WCET yielded by the complex mode w. r. t. the simple mode. The reduction in WCET ranges from 69% to 83% for the 4 tasks.

Future work involves studying the WCET reduction yielded by NUPA within a single task. This entails identifying program segments with single-feasible paths, developing a unified analysis framework that integrates simulation and conventional analysis, and quantifying the overheads of switching between pipeline modes.

4. REPEATABLE EXECUTION CONSTRAINTS FOR OUT-OF-ORDER (RECORDER)

RECORDER enables analysis of multiple control-flow paths on a complex out-of-order pipeline. The key realization is that multiple control-flow paths can be analyzed for an out-of-order pipeline if we can somehow guarantee a single input-independent execution schedule. Accordingly, RECORDER defines a strict set of constraints that guarantee a single input-independent execution schedule on an out-of-order pipeline. If these constraints are met, simulation with an arbitrary (random) input set can be used to derive the execution schedule.

RECORDER specifies the following constraints: (1) the number of dynamic instances of an instruction must be constant across program runs, independent of the input, and (2) a dynamic instance of an instruction must execute at the same time across program runs, independent of the program's input. These constraints guarantee a single input-independent execution schedule on an out-of-order pipeline.

We illustrate the need for the first RECORDER constraint using the example shown in Figure 2. Consider the CFG with three basic blocks shown in Figure 2. For illustration, we show an instruction X in the first basic block and an instruction Y in the middle basic block. Let us assume that the branch at the end of the first basic block is input-dependent. For input 1, the input-dependent branch is taken, i.e., the middle basic block is not executed. Accordingly, the execution schedule for input 1 (shown on the left side of the dotted vertical line) shows instruction X and not instruction Y. On the other hand, for input 2, the branch is not-taken, i.e., the middle block is executed. In this case, both X and Y are executed, as shown in the execution schedule for input 2. Thus, variation in the dynamic instruction stream can lead to different execution schedules.

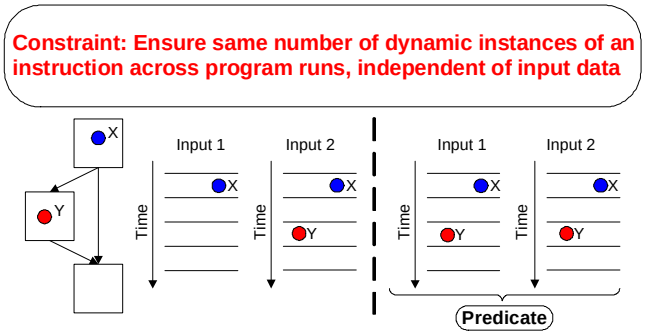

Figure 2. Variability in execution time due to variability in the dynamic instruction stream.

The first constraint stipulates that the number of dynamic instructions must be constant across program runs. This ensures that there is no variation in the dynamic instruction stream. The dynamic scheduler sees a single trace of instructions, that does not change with the program's input. This can be achieved by predicating all input-dependent branches, thereby always executing both paths of these branches. Returning to our example, the input-dependent branch at the end of the first basic block is predicated to satisfy the first constraint. We show the execution schedules for inputs 1 and 2, with the first RECORDER constraint, on the right side of the dotted vertical line in Figure 2. We see that both X and Y are executed for both input 1 and input 2, resulting in an input-independent execution schedule.

Executing both paths of an input-dependent branch may yield multiple potential producers of a value. A later dependent instruction, after the predicated block, may execute at different times depending on which producer is ultimately selected and when that producer executes. Figure 3 illustrates such variations. In this example, we extend the previous example so that there is an instruction Z in the last basic block. Instruction Z depends on either X or Y, depending on whether the input-dependent branch is taken or not-taken, respectively. To satisfy the first RECORDER constraint, the branch is predicated, such that both X and Y are executed, independent of the input. For input 1, Z consumes the value produced by X, i.e., Z depends on X. Assuming that the predicate is already computed, Z can execute as soon as X completes. In this case, Z does not have to wait for Y to complete since it only depends on X. On the other hand, for input 2, Z depends on Y. In this case, Z has to wait until Y completes. Thus, there is timing variability due to uncertainty in who ultimately produces a value (X or Y) and when that producer executes.

The second RECORDER constraint stipulates that there must be no such timing variability. This can be achieved by executing an instruction only after *all* potential producer instructions are executed. In our example, instruction Z has to wait for both its potential producer instructions, X and Y, to complete before it can execute. The right side of the dotted vertical line shows the execution schedules for inputs 1 and 2 when the second RECORDER constraint is enforced. Notice that the execution

schedules for the two inputs are identical when both RECORDER constraints are satisfied.

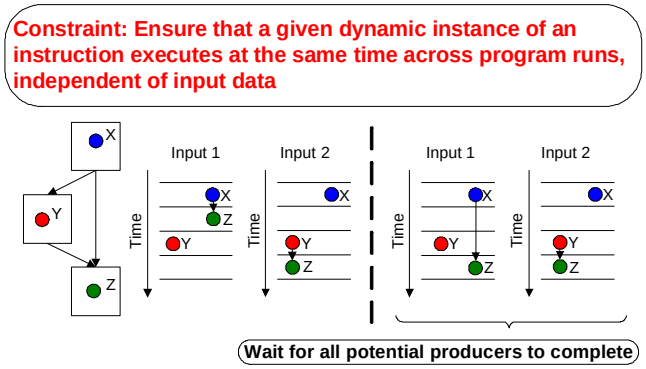

Figure 3. Variability in execution time due to uncertainty in when ultimate producer instruction completes.

The uniqueness of RECORDER is that it only specifies a set of constraints that have to be followed by the dynamic scheduler. The execution schedule does not have to be constructed *a priori*. The execution schedule is determined by the dynamic scheduler at run-time. The dynamic scheduler has the flexibility to schedule instructions out-of-order for high performance. The constraints specified by RECORDER simply guarantee that the dynamic scheduler will always construct the same high-performance execution schedule across program runs, independent of the program's inputs. Thus, instructions can execute out-of-order within a program run, but a specific instance of an instruction always executes at the same time across program runs.

Figure 4 illustrates the benefits yielded by RECORDER. We show an example instruction sequence with oldest instructions shown by a light shade and newest instructions by a dark shade. We also show the actual execution schedules for different program inputs on an out-of-order pipeline that implements RECORDER constraints.

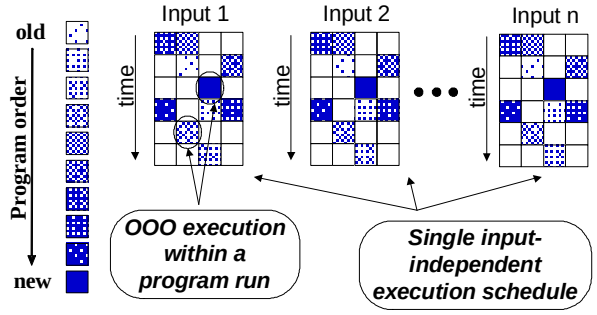

Figure 4. RECORDER benefits.

We see that independent instructions execute out-of-order within a program run. Yet, the execution schedule does not change across program runs, due to meeting RECORDER constraints. This means that the program has a single input-independent execution time, which is both the actual execution time and the worst-case execution time. Now, simulation with an arbitrary (random) input set can be used to determine the program's WCET.

Finally, we show how a previously proposed technique, called Phi-predication [4], satisfies RECORDER constraints. With conventional predication, a branch is converted into a predicate computation and each path (all instructions in that path) is assigned a unique predicate. Phi-predication differs from conventional predication, in that all predicated instructions write their results to the register file and a special "select" instruction (per logical register) at the end of the predicated block is used to select the correct result to be forwarded after the predicated block, based on the outcomes of predicates.

Phi-predication satisfies the first RECORDER constraint since instructions along both paths of a predicated branch are always executed. This guarantees that the number of dynamic instances of an instruction is the same across program runs, independent of the program's input.

Phi-predication uses a select instruction to choose the correct version of a register when there are multiple producers of the register in a predicated region. By introducing the select instruction, a later consumer that uses the register depends on the single select instruction. Moreover, the select instruction depends on all potential producers and it executes only after all the potential producer instructions have executed. This implies that the select instruction and subsequent dependent instructions wait for the slowest potential producer. In this way, the select instruction removes timing variations of a dynamic instance across runs, otherwise caused by input-dependent data-flow, guaranteeing the second RECORDER constraint.

We use a synthetic micro-benchmark to illustrate RECORDER. The benchmark consists of an input-dependent branch inside a loop that is executed 30 times. One path of the input-dependent branch has 15 add instructions and the other path of the branch has 2 add instructions. Figure 5 shows the actual execution times (AETs) of the benchmark on an in-order scalar pipeline with static branch prediction (called *simple*) and an out-of-order 2-issue pipeline with dynamic branch prediction (called *complex*), for 5 different input sets. The third bar shows WCET on *simple* obtained via conventional timing analysis. *Complex* cannot be analyzed, so there is no provably known WCET on *complex*. The last bar shows the AET (also the WCET) on an out-of-order 2-issue pipeline that uses phi-predication (called *complex w/ RECORDER*). The key point to note is that the AET/WCET on *complex w/ RECORDER* remains constant for all 5 inputs, whereas the AETs on *simple* and *complex* vary with the program's input.

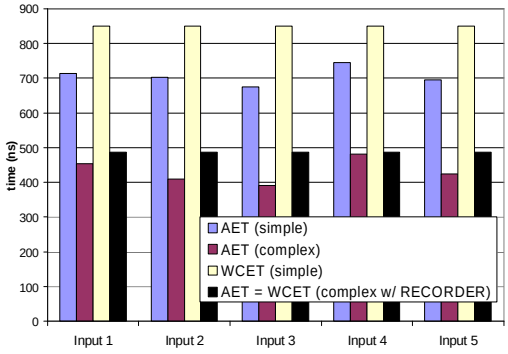

Figure 5. Actual execution times and worst-case execution times for simple and complex with and without RECORDER.

Notice that the AET/WCET on *complex w/ RECORDER* is slightly greater than the AET on *complex*. This is due to the overhead of phi-predication. Nonetheless, *complex w/ RECORDER* yields a 43% reduction in WCET w.r.t. WCET on

simple. Thus, RECORDER guarantees a constant input-independent execution time on an out-of-order pipeline that can be obtained via simulation, with arbitrary inputs, and this WCET is much smaller than the WCET on an in-order pipeline.

5. COMBINING NUPA AND RECORDER

In this section, we describe how NUPA and RECORDER can be used together to improve overall system performance. While NUPA and RECORDER are orthogonal techniques, they are also complementary and work together to improve each other. NUPA benefits from RECORDER because RECORDER enables more program segments to be analyzed in the context of the complex pipeline. On the other hand, the constraints imposed by RECORDER may result in inflated WCETs for some program segments, in spite of the high-performance complex pipeline. Also, RECORDER constraints may result in increased power consumption (for example, both paths of a branch have to be executed, even though only one path is correct). Now, NUPA benefits RECORDER by opting out of the complex pipeline for such problematic segments and analyzing them in the context of the simple pipeline.

We illustrate the benefits of combining NUPA and RECORDER using an example. Consider a program with three program segments A, B, and C. Segment A has a single control-flow path, while segments B and C are composed of multiple control-flow paths. On the left-hand side of Figure 6, we show the WCETs of the three program segments, A, B, and C, for three pipeline models: (1) simple, (2) complex, and (3) complex with RECORDER. All three segments are analyzable for the simple pipeline, and their WCETs in the context of the simple pipeline are shown in light gray. Segment A can be analyzed for the complex pipeline since it has a single control-flow path, and its WCET for complex is shown in dark gray. By virtue of its single control-flow path, segment A implicitly satisfies RECORDER constraints on complex. As a result, segment A has the same WCET for complex and complex with RECORDER (shown in white). Segments B and C cannot be analyzed for the complex pipeline since they are composed of multiple control-flow paths. Hence, WCETs of B and C for complex do not exist and are not shown. But, segments B and C are analyzable for complex with RECORDER. Accordingly, we show WCETs of B and C for complex with RECORDER, in white. In our example, suppose segment B benefits from complex with RECORDER, whereas segment C does not. Accordingly, B's WCET on complex with RECORDER is smaller than its WCET on simple, whereas C's WCET on complex with RECORDER is larger than its WCET on simple.

On the right-hand side of Figure 6, we show the WCET of the whole program, derived using four different timing analysis techniques: (1) conventional, (2) NUPA-only, (3) RECORDER-only, and (4) NUPA with RECORDER. Conventional analysis uses the simple pipeline uniformly and yields the largest WCET. NUPA-only analysis exploits complex for segment A, whereas segments B and C are analyzed in the context of the simple pipeline. NUPA-only analysis yields a WCET that is smaller than the WCET derived by conventional analysis. Third, we show WCET yielded by RECORDER-only analysis, applied to all three segments, A, B, and C. Again, note that segment A does not explicitly need RECORDER since its single control-flow path implicitly satisfies RECORDER constraints. The WCETs of A and B on complex with RECORDER are reduced with respect to their corresponding WCETs on simple, whereas C's WCET is increased with respect to its WCET on simple. Nonetheless, the overall WCET produced by RECORDER-only analysis is still less than the overall WCET produced by conventional analysis. However, in this example, the overall WCET produced by RECORDER-only analysis is greater than the overall WCET produced by NUPA-only analysis. The final technique, NUPA with RECORDER, yields the smallest WCET compared to the WCETs derived by the other three analysis techniques. The flexibility of NUPA with RECORDER allows it to match each program segment with the pipeline model that yields the lowest WCET for that segment, resulting in the best overall WCET. Thus, we see that NUPA and RECORDER can work in combination to yield the lowest WCET.

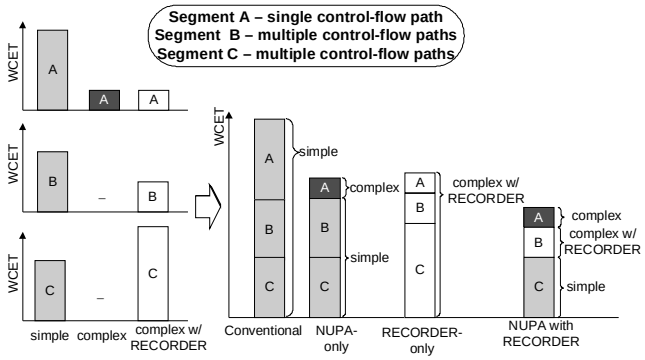

Figure 6. Comparing WCETs yielded by different timing analysis techniques.

6. SUMMARY

We proposed two complementary techniques that enable easy, tight, and safe WCET analysis of complex processors. With NUPA, program segments with a single path are analyzed on a complex pipeline via simulation, and program segments with multiple paths are analyzed on a simple pipeline via conventional static analysis. A reconfigurable processor with dual modes mirrors the hybrid analysis. RECORDER defines constraints that, if met, guarantee a single input-independent execution schedule on an out-of-order pipeline, allowing simulation-based timing analysis with arbitrary inputs even for program segments with variable control-flow. NUPA and RECORDER work hand-in-hand to capitalize on the performance of contemporary microarchitectures in hard-real-time systems.

7. REFERENCES

[1] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller, "Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems", *Proc. Int. Sym. on Computer Architecture*, 2003.

[2] R. Ernst and W. Ye, "Embedded Program Timing Analysis Based on Path Clustering and Architecture Classification", *Proc. Int. Conf. on CAD*, 1997.

[3] P. Puschner, A. Burns. "Writing Temporally Predictable Code". *Proc. 7th Int. Workshop on Object-Oriented Real-Time Dependable Systems*, 2002.

[4] W. Chuang, B. Calder, J. Ferrante. "Phi-Predication for Light-Weight If-Conversion", *Proc. Int. Sym. on Code Generation and Optimization*, 2003.