Efficiently Tolerating Timing Violations in Pipelined Microprocessors

Koushik Chakraborty Brennan Cozzens Sanghamitra Roy Dean M. Ancajas

USU BRIDGE LAB, Electrical and Computer Engineering, Utah State University
{koushik.chakraborty, sanghamitra.roy}@usu.edu

ABSTRACT

Early prediction of an upcoming timing violation presents a tremendous opportunity to mask the performance overhead of tolerating these faults. In this paper, we explore several techniques for optimizing instruction scheduling in an Out-of-Order pipeline, exploiting this new perspective in robust system design. Compared to recently proposed stall based techniques for tolerating predictable timing violations, we demonstrate a massive reduction in performance overhead, while supporting correct execution in faulty environments (64–97% across different benchmarks).

Categories and Subject Descriptors

B.8.1 [**Hardware**]: Reliability, Testing and Fault Tolerance

General Terms

Reliability

Keywords

Timing Faults, Path Sensitization, Instruction Scheduling.

1. INTRODUCTION

Growing unreliability in electronic systems is reshaping the design approaches of the computing world. In this domain, timing violations—an artifact of rapid technology scaling—embody a central reliability challenge [1, 2]. Guided by a combined effect of static (process variation and wearout) and temporal (thermal, voltage or utilization) variation, timing violations can occur sporadically [1, 2]. Consequently, runtime error detection and correction techniques have been a topic of major research in recent years [3–7]. Existing works in this area either provide a very high fault coverage at the expense of a large performance overhead, or provide a poor fault coverage with a low performance penalty [3, 6–11].

In this paper, we demonstrate that it is possible to approach the performance of fault-free execution, while tolerating timing errors in an Out-of-Order (OoO) microprocessor pipeline. We establish a foundation for low-overhead timing-error tolerance using a violation aware instruction scheduling framework. This unique framework is based on the recently observed predictability of timing errors from specific instructions causing them [12, 13]. We can use the instruction Program Counter (PC), a unique instruction identifier, for predicting an upcoming timing violation, several clock cycles in advance. While tolerating an unexpected timing error entails a large performance overhead in pipelined architectures [3], we observe that scheduling an instruction with a predictable timing error becomes logically equivalent to a variable latency operation.

To exploit this property, our proposed violation aware instruction scheduling framework ensures two fundamental principles: (a) the faulty instruction occupies one additional cycle in the same pipe stage, and no new input is fed in the resources occupied by that instruction during this time; and (b) the dependent instructions behind the faulty instruction are held back by one extra cycle. Our proposed techniques can thus mask the penalty of timing violations from the system performance by confining the error overhead to the faulty instruction and its dependents only. Our proposed technique avoids stalling the whole pipeline as was done in recent works [12, 13], thereby marginalizing the system level overhead of tolerating timing violations.

Enabled by our violation aware scheduling techniques, microprocessors can operate at a tighter frequency, where predictable errors frequently occur and are tolerated with minimal performance loss. The main contributions of our paper are outlined next.

- We propose a series of low overhead micro-architectural enhancements coupled with instruction scheduling techniques to drastically limit the performance overhead of predictable timing faults (Sections 2, 3).
- Using a rigorous circuit-architectural simulation (Sections 4 and 5), combining synthesized hardware with full system simulation, we demonstrate dramatic reduction in the performance overhead from faulty execution (64–97%) compared to stall based schemes [12, 13]. Our proposed schemes have negligible power/area overhead giving an energy-efficient alternative for robust pipelines (Section S3).
- We study the locality of sensitized paths from multiple dynamic instances of a given instruction in several microprocessor structural blocks. Our analysis, employing the Fabscalar infrastructure [14], demonstrates the predictability of timing faults in these components (Section S1).

2. ROBUST PIPELINE DESIGN OVERVIEW

In this section, we present an overview of our proposed timing error tolerant Out-of-Order (OoO) pipelined microprocessor. The goal of our proposed techniques is to approach the performance of fault-free execution while tolerating timing errors. We outline an overview of our design in Section 2.1 and summarize how timing errors are handled in various stages of the pipeline (Section 2.2).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2013, May 29–June 07, Austin, TX, USA
Copyright 2013 ACM 978-1-4503-2071-9/13/05 ...$15.00.

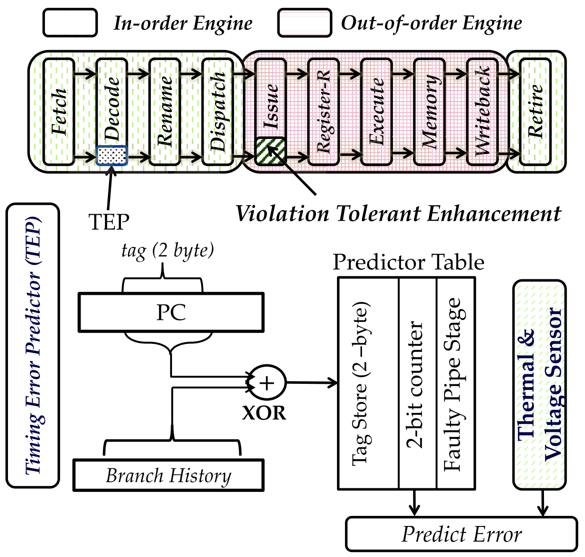


Figure 1: An Overview of our proposed techniques showing key enhancements. The decode stage is augmented with a Timing Error Predictor (TEP). This predictor information is propagated in subsequent pipe stages. The Issue stage is enhanced for violation tolerant scheduling.

2.1 Pipeline Overview

Figure 1 shows the overview of our pipelined micro-architecture. An OoO processor typically consists of a front end spanning from *Fetch* to *Dispatch*, where instructions proceed in-order, the OoO core engine spanning from *Issue* to *Writeback*, and an in-order *Retire* stage. An individual stage in this pipeline may also span across multiple clock cycles [14].

We augment this pipeline with several micro-architectural enhancements. The decode stage is enhanced with a Timing Error Predictor (TEP), which dynamically learns and predicts timing errors in the pipeline from a decoded instruction PC. The TEP is accessed in parallel to decode avoiding any impact on the critical path. The TEP prediction is subsequently propagated with the instruction meta-data as it traverses through various pipeline stages. Subsequently, the issue stage recognizes an instruction with a predicted timing error and activates violation aware instruction scheduling. For this purpose, the issue stage micro-architecture is augmented with a Violation Tolerant Enhancement (VTE) detailed in Section 3. If an instruction meta-data does not indicate a timing violation, instructions proceed normally. We next discuss our TEP design, and then briefly outline how timing error mis-predictions are handled.

2.1.1 Timing Error Predictor

Our TEP combines features from the Most Recent Entry (MRE) predictor proposed by Xin et al. with the Timing Violation Predictor (TVP) proposed by Roy et al. [12, 13]. Figure 1 shows our TEP design. Each entry in the predictor table contains a 2 byte tag obtained from the PC. The entries in the table are indexed using a combination of bits in the PC and the recent branch outcomes. A 2-bit saturating counter in each entry keeps track of the potential of a timing error in the system. A non-zero value in the saturating counter indicates a possible timing violation. We also keep track of the faulty pipe stage associated with an error causing instruction. The prediction also considers favorable conditions for timing errors through the use of thermal and voltage sensors.

2.1.2 Handling Mis-prediction

If an instruction incurs a timing violation without early prediction, an error recovery is triggered using instruction replay, similar to *Razor* [15]. This recovery corrects a fault that is not handled by our violation aware scheduling framework. Instruction replays are rare, but incur a large performance overhead.

2.2 Tolerating Timing Violations in Specific Pipe Stages

Timing errors may happen in the in-order engine or the OoO engine of the processor. Our experiments, as well as, recent works have indicated that the likelihood of timing errors is significantly more in the OoO engine [16]. Therefore, our proposed techniques are primarily focused on efficiently tolerating timing errors in the OoO core. However, for the sake of completeness, we outline how we handle errors in the in-order engine next.

In-order Engine: The violation aware scheduling framework is not applicable to the in-order part of the pipeline. For the rename, dispatch and retire stages, we use the predicted violation from the TEP to enable a stall signal at the appropriate stage. This stall signal allows the faulty pipe stage to complete in two clock cycles, while the input to all other stages are recirculated to avoid forward flow of instructions during that cycle. The stall signal can be enabled using existing circuitry in modern microprocessors with minimal modification. The TEP cannot be used to mitigate timing violations in the fetch and decode stages of the pipeline. Any violations in these two stages are mitigated using instruction replay with our error recovery circuitry discussed in Section 2.1.2. Such replays are however rare, as the fetch and decode stages have substantially lower fluctuations of temperature and voltage making timing violations rare [17].

OoO Engine: We use our proposed timing violation aware instruction scheduling framework for tolerating timing errors in the OoO engine, discussed next.

.

3. VIOLATION AWARE SCHEDULING

In this section, we describe our violation tolerant enhancement (VTE) and scheduling algorithms in a pipelined OoO microprocessor. Our goal is to efficiently tolerate timing violations in the OoO engine shown in Figure 1, radically improving upon stall based techniques [12, 13].

3.1 Violation Aware Scheduler Overview

From an instruction scheduling perspective, a pipe stage execution with a timing violation becomes equivalent to a variable latency operation *when that violation can be predicted early*. Consequently, we can suitably alter the instruction scheduling in a manner such that: (a) the faulty instruction occupies one additional cycle in the same pipe stage, and no new input is fed in the resources occupied by that instruction during this time; and (b) the dependent instructions behind the faulty instruction are held back by one extra cycle. At its core, these *scheduling features* require modification to the pipe resource management and the communication of dependency between two instructions (detailed in Section 3.2). Eventual impact of these corrective measures on the processor performance is determined by the existing *architectural slack* of the faulty instruction [18] (e.g., increased latency on some instructions may have negligible impact on the system performance).

3.2 Violation Tolerant Enhancements

To ensure correct execution with timing violations, we need to make micro-architectural modifications in the scheduler logic in the issue stage. Other stages within the out-of-order engine require supporting modifications. First, we describe the VTE in the Issue stage. Then we describe how predictable timing violations are tolerated in all stages within the OoO engine (Section 3.3). Three major aspects of the VTE are: (a) Issue Queue Entry; (b) Tag Broadcast Logic and (c) Issue Slot Management.

3.2.1 Issue Queue Alteration

The issue queue entries are augmented to include a single-bit that indicates the *fault prediction* of an instruction. Furthermore, another field indicates the faulty pipe stage, so that the pipe stage logic is modified when that instruction enters the faulty stage. Combined together, a 4-bit field is sufficient to encode the error prediction information for each instruction.

3.2.2 Tag Broadcast Logic

When an instruction is scheduled, engaging the Register Read, Execute and Memory stages, the scheduler logic keeps track of its expected completion time. Subsequently, in the cycle the instruction completes, the instruction tag is broadcast to the issue queue. Waiting instructions then perform a tag match with this result tag, to evaluate if their operands are ready. Based on whether the completion is triggered from the Memory stage (load instructions) or Execute stage (ALU instructions), we use a countdown to keep track of its completion time. In case of a faulty instruction, we increment the completion counter, based on its expected delay. Thus, the tag broadcast is delayed by one cycle.

3.2.3 Issue Slot Management

In each cycle, the issue stage in the OoO engine prepares a packet consisting of several instructions (equal to the pipeline width W), which is then propagated through the later stages of the pipeline. We denote the position of a particular instruction in this packet as an issue slot in our description.

When a faulty instruction is issued, the issue slot occupied by that instruction must be managed carefully. This is necessary to avoid sending a new instruction in the same slot to the faulty stage before the faulty instruction has sufficient time to complete its computation. To accomplish this task, we keep track of the issue slot occupied by a faulty instruction. In the subsequent cycle, we freeze the slot to disallow issuing another instruction behind the faulty instruction. We next discuss, how each pipe stage in the OoO engine tolerates timing violations.

3.3 Tolerating Timing Errors in the OoO Pipe Stages

3.3.1 Issue

Issue can have multiple pipe stages. However, the wakeup/select stage inside the issue is particularly prone to timing errors due to the use of content addressable memory (CAM) logic in the wakeup. In our experiments, also corroborated by others [16], we find that almost all timing errors happen in the wakeup/select stage. The issue stage is responsible for handling timing errors in the other OoO pipe stages. However, a timing error in the issue itself can cause a pipeline deadlock when back-end errors are relying on correct operation of the issue.

To avoid such a pipeline deadlock, we adopt a low-complexity technique that trades off marginal performance loss for complexity reduction. After an instruction with a predictable timing error in the issue is inserted in the *issue queue* from the dispatch stage, we track the functional unit or memory port where the faulty instruction will be scheduled. Once this faulty instruction is scheduled, we freeze the corresponding issue slot for the functional unit or memory port in the subsequent cycle. Consequently, when this faulty instruction broadcasts its tag, the input to the wakeup select lane will remain steady for two cycles, thereby providing sufficient time to complete the logic computation.

3.3.2 Register Read

When an instruction has a predictable timing error in the register read stage, the issue queue blocks the respective register read port, where the faulty instruction is assigned, for one additional cycle. This blocking allows the instruction to complete register read in two cycles, and avoids using the read port in the next cycle after the faulty instruction enters the register read stage. The scheduling cycle for the execute/memory stage for this faulty instruction is adjusted to handle this additional delay from the register read stage.

3.3.3 Execute

The execute stage is composed of various functional units. The key to functional unit management is to ensure that the *instructions are issued to functional units when they are ready to process new instructions*. To keep track of this information, we use a *Functional Unit State Register (FUSR)*. Each bit in the FUSR keeps track of one functional unit, and indicates if a new instruction can be issued to that unit in the next cycle. We now discuss two major classes of functional units based on their completion delay: single cycle and multi-cycle.

Single-Cycle Latency: Functional units with a single cycle latency can process new instructions in every cycle. However, when a faulty instruction is scheduled, its FUSR bit is turned off for one cycle to disallow issuing a new instruction in the next cycle.

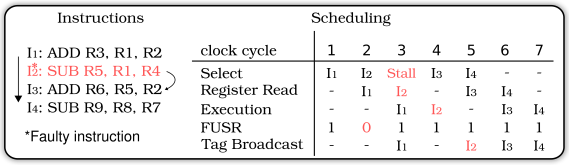
Multi-cycle Latency: A multi-cycle functional unit may or may not be pipelined. In the case it is not pipelined, the FUSR is adjusted to indicate the busy state for one extra cycle beyond its expected completion time. A fully pipelined functional unit can process new instructions every cycle in the absence of timing errors. However, a timing error can happen in any of the internal pipeline stages. To effectively handle these multi-cycle pipelined units, we temporarily avoid issuing new instructions behind a faulty one. We resume issuing new instructions to that unit only after the faulty instruction completes. This approach is agnostic of the exact error pipe stage within the multi-cycle execution unit, thereby saving design complexity at the cost of a marginal performance loss.

3.3.4 Memory

Similar to the issue, the memory stage is also susceptible to errors due to the presence of CAM logic in the load-store queue. In particular, when the CAM search results in several tag matches, we observe additional delay in this stage, potentially causing timing errors. When the issue queue schedules a predictable faulty instruction to the memory, it estimates the cycle when the CAM match will be performed. Based on this estimation, the issue queue avoids issuing a load/store instruction behind the faulty one to prevent another CAM match in the cycle right after the faulty instruction. Consequently, the faulty instruction can continue to do the CAM match for two cycles. The writeback stage of this faulty instruction is delayed in the memory stage by one cycle to preserve correct execution.

3.3.5 Writeback

The writeback stage is relatively less susceptible to errors, compared to other stages discussed above. However, to achieve fault coverage in the entire OoO engine, we propose some enhancements to the Writeback stage. Every cycle, this stage receives *W* pack-



Instructions

I_1: ADD R3, R1, R2
I_2^*: SUB R5, R1, R4
I_3: ADD R6, R5, R2
I_4: SUB R9, R8, R7

*Faulty instruction

Scheduling

clock cycle	1	2	3	4	5	6	7
Select	I_1	I_2	Stall	I_3	I_4	-	-
Register Read	-	I_1	I_2	-	I_3	I_4	-
Execution	-	-	I_1	I_2	-	I_3	I_4
FUSR	1	0	1	1	1	1	1
Tag Broadcast	-	-	I_1	-	I_2	I_3	I_4

Figure 2: An example showing the scheduler logic modification.

ets, each of which consist of information necessary to complete the writeback of an in-flight instruction (W is the issue width). If one of the instructions has a potential timing error in the writeback, then its corresponding input slot is frozen by the issue queue in the next cycle, allowing the input to recirculate.

3.4 An Illustrative Example

Figure 2 shows an operational example of scheduling instructions around a faulty instruction. We assume a functional unit with one-cycle completion time, so that new instructions can be scheduled in every cycle, while scheduled instructions complete in the same cycle. The instruction I_2 is predicted to have a fault in the execution unit. This instruction is selected in cycle 2, to be executed on the functional unit on cycle 4. However, as it is faulty, it takes one additional cycle. Moreover, as no new instruction can be scheduled on cycle 5 on that functional unit, the FUSR is marked 0 at the end of cycle 2 to avoid selecting a new instruction for that functional unit in cycle 3 (representing issue slot freezing discussed in Section 3.2.3). The tag broadcast logic is delayed by one cycle, so that dependent instruction I_3 is held back for one cycle.

3.5 Violation Aware Scheduling Algorithms

Beyond ensuring correct execution in the presence of predictable timing violations, the next design issue pertains the selection priority of instructions with operands ready. We explore three different algorithms for instruction scheduling, all of which confine the penalty to a faulty instruction and its dependents, and aim to minimize the system level performance overhead of a timing fault:

- Age based selection (ABS)
- Faulty First Selection (FFS)
- Criticality Driven Selection (CDS)

The age based policy, ABS, uses a timestamp—implemented using a 6-bit module-64 counter—to select instructions to schedule, among those that are operand ready. The faulty first policy, FFS, attempts to schedule instructions with faults early, so as to release their dependent instructions sooner. The criticality driven policy dynamically estimates the criticality of a predictable faulty instruction (detailed in Section 3.5.2), and attempts to eagerly select those instructions that have a higher criticality. We next outline our proposed issue queue modifications to implement these policies, and subsequently discuss our approach for dynamic criticality estimation.

3.5.1 Selection Logic Enhancement (SLE)

Figure 3 shows the implementation of the SLE for the proposed priority schemes discussed above. Each issue queue entry keeps track of three major aspects necessary for selection: (i) operand ready; (ii) timestamp; and (iii) a 4-bit field indicating the fault prediction (Section 3.2.1) and criticality of an instruction. The faulty bit is also used to manage functional units and the tag broadcast logic, as discussed in Section 3.2. All instructions with operand ready, bids for selection (Figure 3). The ABS policy sets the grant

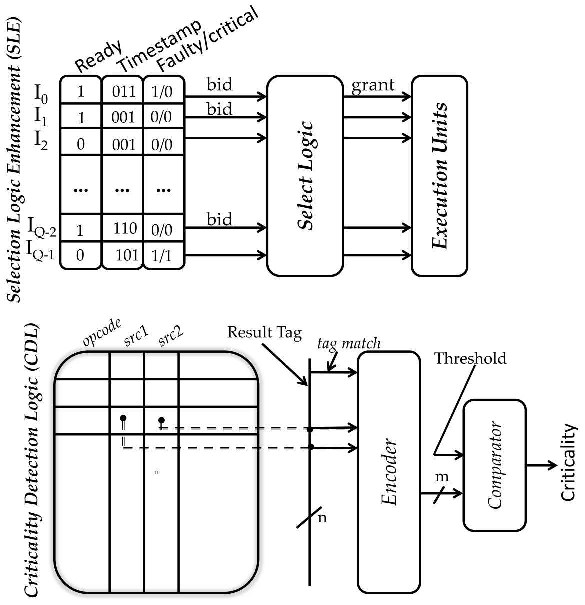


Figure 3: SLE and CDL Implementation.

line for the instructions with the lowest timestamp (oldest instructions). The FFS policy sets the grant line for instructions with faulty bit set. When none of the instructions are faulty, then it uses the timestamp to select instructions (similar to ABS). The CDS policy eagerly selects faulty instructions that are expected to be critical. Again, similar to FFS, if no such instructions (faulty and critical) exist, then it uses the timestamp.

3.5.2 Criticality Detection Logic (CDL)

Precisely estimating the criticality of an instruction in hardware is challenging, as the hardware has limited information about the dynamic data flow of a program [18]. Instead, we use a low-complexity technique to estimate the instruction criticality by tracking the number of dependent instructions behind a given instruction in the issue queue. Figure 3 shows the implementation of our proposed scheme in a reservation station. When an instruction broadcasts its result tag, we track the number of tag matches in the reservation station. These tag matches are fed to an encoder, and then compared with a predefined *Criticality Threshold (CT)*. This CT dictates the minimum number of dependent instructions that must be present in the issue queue to consider a given instruction to be critical. Once we determine this instruction criticality, we store this information with the timing error predictor (Section 2.1.1). In our experiments, we find that a CT of *8* gives the best outcome.

4. METHODOLOGY

In this section, we describe our extensive circuit-architectural methodology for performance tradeoff analysis of our proposed techniques.

4.1 Circuit Implementation

We implement our proposed scheduling techniques within the Fabscalar infrastructure [14]. For the purpose of this paper, we use the *Core-1* configuration, which represents an out-of-order pipeline capable of fetching, issuing and committing 4 instructions each cycle. The pipeline has single-cycle (e.g. simple ALU) as well as multi-cycle (e.g. complex ALU) functional units. The misprediction loop for this pipeline is 10 stages, spanning fetch to execute. We synthesize our implementation with the Synopsys Design Compiler using a 45nm FreePDK library. Energy results are gathered by combining architectural usage information with power characteristics from the synthesized hardware.

4.2 Architectural Simulation

We use full-system simulation built on top of WindRiver SIMICS [19]. We use our own detailed timing model to enforce the timing characteristics of a 4-wide out-of-order microprocessor, identical to the Core-1 configuration mentioned above. The core uses a two-level cache hierarchy where L1 (32KB 4-way split Instruction and Data) has a single cycle latency, while the 16-way 8MB L2 and the main memory are accessed in 25 and 240 cycles, respectively. We use the TEP as our predictor design (Section 2.1.1). For both fault-free execution and Error Padding scheme (discussed in Section 5) [13], we use the age based instruction selection policy. We use several SPEC CPU2006 benchmarks, and focus our architectural simulation on representative phases extracted using the SimPoint toolset [20]. Each phase corresponds to 1 million committed instructions.

4.3 Fault Simulation

To simulate timing faults, we embed gate delay information in the architectural simulation. The effect of process variation and aging on the circuit timing is obtained by our in-house statistical timing tool that uses SPICE characterized gate delay distributions [21]. To model process variation, we assume that the transistor length, width and oxide thickness behave as Gaussian distributions with $\pm 20\%$ deviation across the nominal values [1, 22].

For the purpose of this paper, we focus on timing violations in the OoO engine of the processor, spanning from *Issue* to *Writeback*. Together, these stages comprise the heart of the control and datapath in a pipelined microprocessor, and also contain the timing critical stages in the microprocessor [16].

Depending on the program input, different instructions incur different delays based on the delays in individual gates in the sensitized paths. We alter the supply voltage to create two different faulty environments: high fault rate (0.97V), and low fault rate (1.04V). Faults are assumed to occur when the 95% confidence interval of the stage delay exceeds the cycle time ($\mu + 2\sigma$). The baseline machines have zero fault rate when executing at 1.1V supply voltage. Most of the timing violations are accurately predicted and tolerated with one of the comparative schemes. However, when timing faults occur without early prediction, we initiate error recovery using instruction replay, similar to Razor [3].

5. EXPERIMENTAL RESULTS

In this section, we present experimental results of our proposed schemes for optimizing scheduling around predictably faulty instructions. Our goal is to study the power-performance overhead incurred *during* faulty execution.

Comparative Schemes: We study the following schemes:

- **Razor:** This scheme fires an instruction replay for all errors in the system [3].
- **Error Padding (EP):** This is our baseline scheme that introduces stall cycles for predicted errors, similar to [12, 13].
- **ABS, FFS, CDS:** These are our proposed schemes for violation aware scheduling described in Section 3.5.

5.1 Fault Rates

Depending on specific paths sensitized during program execution, different benchmark programs exhibit different fault rates while

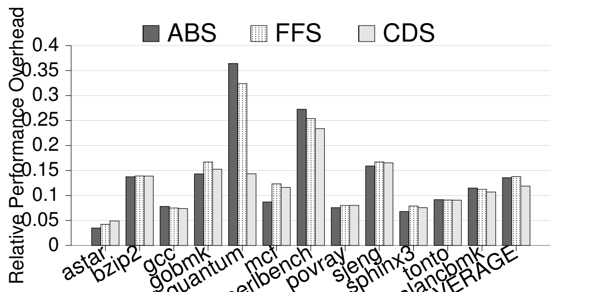


Figure 4: Performance Overhead Comparison During Faulty Execution at low fault rate (1.04V) normalized to EP [12, 13]. (Lower is better.)

operating at the same supply voltage. In Table 1, we report the average fault rates seen in the OoO engine, collectively, when operating under 1.04V and 0.97V, respectively. We also report the performance overhead over fault-free execution seen with Razor [3] and EP schemes. Overheads are shown as a tuple, representing the percentage of performance and energy efficiency degradation, respectively. Performance is estimated using *Instruction per cycle (IPC)*, while energy efficiency is estimated using energy-delay product. Certain benchmarks like *sjeng*, with higher inherent instruction level parallelism, show greater susceptibility to timing violations. On the other hand, benchmarks like *libquantum*, with greater data stalls, show substantially lower performance impact from occasional timing violations. As Razor has substantially higher overhead than EP for performance and energy efficiency, we provide all our subsequent results normalized to the EP scheme.

5.2 Performance Overhead Comparison

Figures 4 and 5 present the relative performance and ED (Energy-Delay) overhead comparison of our proposed schemes, normalized to the baseline *Error Padding* (EP) scheme during lower fault rate. All of our schemes are remarkably effective in marginalizing the performance overhead during timing violations when compared with the baseline. During a lower fault rate (V_{DD} = 1.04V), on an average our schemes reduce the performance overheads by 87% compared to *Error Padding* (Figure 4). Likewise, on an average, our schemes reduce the ED (Energy Delay) overhead by 82% compared to EP (Figure 5). For example, in *astar*, ABS is able to dramatically erase the performance under faulty environment by 97%, delivering performance similar to fault-free execution. ABS also shows similar improvements in ED overhead. On the other hand, in *libquantum*, CDS is particularly effective as it can eliminate 86% of the timing violation penalty, compared to 64% in ABS. Our low complexity criticality assessment is highly effective in the particular data flow pattern in *libquantum*, thereby resulting in a substantial performance advantage. Our schemes are also highly effective during high fault rates (results in the higher fault rate (V_{DD} = 0.97V) are presented in Section S2).

6. RELATED WORK

Recent works in tackling timing faults in pipelined microprocessors can be broadly classified into three groups: reactive, proactive, and predictive. Reactive techniques primarily focus on precise fault detection. Once detected, they ensure correct execution through costly instruction replay [3, 15]. Despite this large performance overhead for error correction, these techniques are often necessary to achieve full fault coverage. Proactive techniques aim to reduce the correction overhead by taking corrective action just before the

Benchmark	Fault-Free IPC	V_{DD}=0.97V			V_{DD}=1.04V		
		FR (%)	Razor Overhead	EP Overhead	FR (%)	Razor Overhead	EP Overhead
astar	0.69	6.74	(31.2, 45.6)	(5.17,6.45)	2.01	(10.2,14.6)	(1.29,1.7)
bzip2	1.48	8.92	(43.2, 56.8)	(12.35,16.5)	2.24	(17.4,25.6)	(3.1,3.7)
gcc	1.34	8.43	(47.2, 61.3)	(8.57,10.3)	1.5	(19.4,29.6)	(2.14,2.6)
gobmk	1.68	8.64	(47.3, 53.3)	(12.65,16.3)	2.16	(18.2,24.5)	(3.16,3.95)
libquantum	0.51	10.54	(25.3, 32.5)	(4.5,5.7)	2.1	(6.8, 10.2)	(1.12, 1.5)
mcf	0.34	6.45	(30.1, 42.3)	(1.96,2.8)	1.73	(9.5,12.6)	(0.49, 0.85)
perlbench	1.31	7.21	(45.7, 54.7)	(6.52,7.1)	1.8	(15.6,21.2)	(1.63, 2.1)
povray	1.941	6.31	(51.2, 75.4)	(7.58,9.1)	1.57	(24.5, 32.5)	(1.89, 2.25)
sjeng	1.93	9.19	(58.6, 72.5)	(15.19,17.8)	2.29	(23.5,29.8)	(3.79, 4.83)
sphinx3	1.30	6.95	(52.5, 67.4)	(5.45,5.9)	1.73	(17.2, 22.5)	(1.36, 1.78)
tonto	1.41	5.59	(45.6, 65.7)	(5.04,6.5)	1.39	(16.5, 21.4)	(1.25, 2.6)
xalancbmk	0.51	7.95	(34.5, 45.2)	(3.093.8)	1.99	(12.5, 15.6)	(0.77, 1.02)

Table 1: Benchmark Fault Rates (FR), and fault tolerance overhead employing Razor [3] and Error Padding [12, 13].

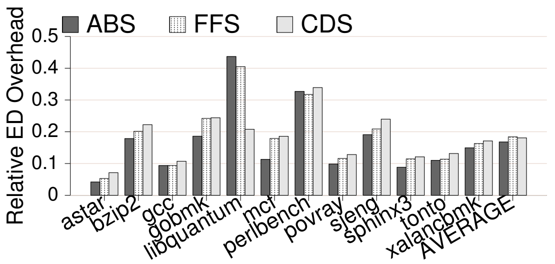


Figure 5: ED Overhead Comparison During Faulty Execution at low fault rate (1.04V) normalized to EP. (Lower is better.)

fault occurrence: in the same clock cycle where timing faults are about to happen [11]. Upcoming faults are anticipated using timing sensors. However, a lack of sufficient time limits the scope of corrective techniques, and hurts their fault coverage [12]. Recent work on predictive techniques aim to predict an upcoming timing fault, several clock cycles in advance [12, 13]. However, neither of these works exploit the immense potential of predicting timing faults. In this work, we explore micro-architectural techniques to radically marginalize the overhead from timing faults, and demonstrate massive improvements over the existing techniques based on early prediction.

7. CONCLUSION

Predicting timing violations offers a tremendous leverage in marginalizing the performance overhead of tolerating recurring timing violations. In this paper, we propose three techniques to schedule instructions with predicted timing violations. Our goal is to confine the performance impact of timing faults on the faulty instructions, while eliminating its impact on other independent instructions. Compared to recently proposed techniques to tolerate predictable timing violations, our proposed schemes dramatically reduce the performance overhead by 64–97% across different faulty environments, while reducing the ED overhead by 58-96%.

Acknowledgments

This work was supported in part by National Science Foundation grants CNS-1117425 and CAREER-1253024, and donation from the Micron Foundation.

8. REFERENCES

[1] S. Sarangi, B. Greskamp and others,“Varius: A model of process variation and resulting timing errors for microarchitects,” *IEEE Transactions on Semiconductor Manufacturing*, vol. 21, no. 1, pp. 3 –13, 2008.

[2] S. Pan, Y. Hu, and X. Li, “Ivf: Characterizing the vulnerability of microprocessor structures to intermittent faults,” in *Proc. of DATE*, pp. 238–243, 2010.

[3] S. Das, C. Tokunaga and others,“Razorii: In situ error detection and correction for pvt and ser tolerance,” *J. of Solid-State Circ.*, vol. 44, pp. 32–48, jan. 2009.

[4] S. Das, D. Roberts and others,“A self-tuning dvs processor using delay-error detection and correction,” *Solid-State Circuits, IEEE Journal of*, vol. 41, pp. 792 – 804, april 2006.

[5] K. Bowman, J. Tschanz and others,“Circuit techniques for dynamic variation tolerance,” in *Proc. of DAC*, pp. 4–7, 2009.

[6] A. B. Kahng, S. Kang and others,“Designing a processor from the ground up to allow voltage/reliability tradeoffs,” in *HPCA*, pp. 1–11, 2010.

[7] A. B. Kahng, S. Kang and others,“Recovery-driven design: a power minimization methodology for error-tolerant processor modules,” in *Proc. of DAC*, pp. 825–830, 2010.

[8] B. Greskamp, L. Wan and others,“Blueshift: Designing processors for timing speculation from the ground up,” in *HPCA*, pp. 213–224, 2009.

[9] A. Tiwari, S. R. Sarangi, and J. Torrellas, “Recycle: pipeline adaptation to tolerate process variation,” in *Proc. of ISCA*, pp. 323–334, 2007.

[10] J. Long and S. O. Memik, “Automated design of self-adjusting pipelines,” in *Proc. of DAC*, pp. 211–216, 2008.

[11] M. Ghasemazar and M. Pedram, “Minimizing the energy cost of throughput in a linear pipeline by opportunistic time borrowing,” in *Proc. of ICCAD*, pp. 155–160, 2008.

[12] S. Roy and K. Chakraborty, “Predicting timing violations through instruction level path sensitization analysis,” in *Proc. of DAC*, pp. 1074–1081, 2012.

[13] J. Xin and R. Joseph, “Identifying and predicting timing-critical instructions to boost timing speculation,” in *Proc. of MICRO*, pp. 128–139, 2011.

[14] N. K. Choudhary, S. V. Wadhavkar and others,“Fabscalar: composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template,” in *Proc. of ISCA*, pp. 11–22, 2011.

[15] D. Ernst, N. S. Kim and others,“Razor: A low-power pipeline based on circuit-level timing speculation,” in *Proc. of MICRO*, pp. 7–18, 2003.

[16] J. Sartori and R. Kumar, “Compiling for energy efficiency on timing speculative processors,” in *Proc. of DAC*, pp. 1301–1308, 2012.

[17] F. J. Mesa-Martinez, J. Nayfach-Battilana, and J. Renau, “Power model validation through thermal measurements,” in *Proc. of ISCA*, pp. 302–311, 2007.

[18] B. A. Fields, R. Bodík, and M. D. Hill, “Slack: Maximizing performance under technological constraints,” in *Proc. of ISCA*, pp. 48–58, 2002.

[19] P. S. Magnusson, M. Christensson and others,“Simics: A full system simulation platform,” *IEEE Computer*, vol. 35, pp. 50–58, Feb 2002.

[20] T. Sherwood, E. Perelman, and B. Calder, “Basic block distribution analysis to find periodic behavior and simulation points in applications,” in *PACT*, pp. 3–14, 2001.

[21] S. Kothawade, K. Chakraborty and others,“Analysis of intermittent timing fault vulnerability,” *Microelectronics Reliability*, vol. 52, pp. 1515–1522, July 2012.

[22] W. Zhao, F. Liu and others,“Rigorous extraction of process variations for 65-nm cmos design,” *IEEE Transactions on Semiconductor Manufacturing*, vol. 22, no. 1, pp. 196 –203, 2009.

Supplemental Materials

S1. INSTRUCTION LEVEL PREDICTABILITY OF TIMING VIOLATIONS

Timing violation predictability, has been studied by two recent works [12, 13]. In this section, we verify this intriguing property using a more elaborate and detailed methodology. We next present our cross-layer analysis combining information from the application, architecture and circuit layers. Our goal is to show the underlying causes of specific instructions causing repeated timing violations—the primary reason behind timing violation predictability. Subsequently, we identify the tremendous opportunity of tolerating a timing violation using violation aware instruction scheduling techniques that hasn't been exploited by prior works.

S1.1 Underlying Cause Behind Timing Violation Predictability

When a certain static instruction executes repeatedly, its many dynamic instances sensitize strikingly similar logic paths in a given circuit block. Consequently, there is a high commonality in the critical paths sensitized by many dynamic instances of a static PC. Hence, if a certain PC causes a timing violation in a circuit block, future occurrences of that PC is highly likely to cause timing violations under identical temperature and voltage conditions. This phenomenon helps us to use the instruction PC to predict an upcoming timing violation in a pipe stage, several clock cycles early.

S1.2 Cross-Layer Methodology

Figure 6 presents an overview of our extensive cross-layer methodology. We use the Fabscalar system design environment that allows us to create and verify superscalar microprocessor cores [14]. Using Fabscalar, we extract synthesizable RTL designs of a few critical components of a microprocessor core (details in Section S1.2.2). The Fabscalar environment helps us to combine architectural simulation of real programs with gate level logic simulation using the Cadence NC-Verilog functional verification tool. We simulate the execution of six SPEC2000 integer benchmarks on each core component to obtain the inputs corresponding to specific instructions.

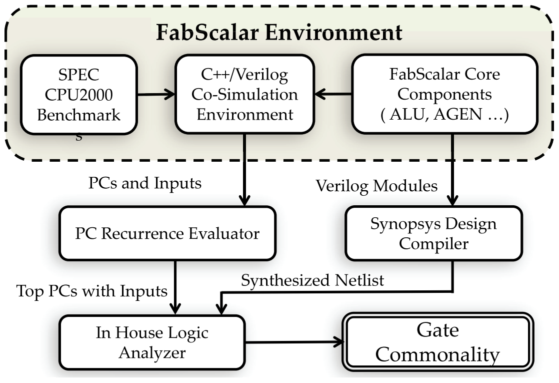


Figure 6: Cross-layer Methodology.

We study the timing violation predictability of instructions, each identified by a unique Program Counter (PC). Along with each PC, we collect its respective inputs for each benchmark. For each PC, we also identify the preceding instruction PC that sets the internal logic state of a microprocessor component. Finally, our in house logic analyzer combines inputs from several repeated instances of a PC with a synthesized microprocessor component.

Module	Issue Queue Select	ALU	AGEN	Forward Check
# Gates	189	4728	491	428
Logic Depth	33	46	43	15

Table 3: Details of Synthesized Processor Components.

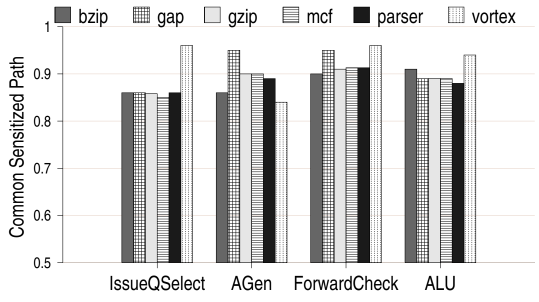


Figure 7: Commonality in sensitized paths in four components of a microprocessor core.

Commonality Estimation: We estimate the gate level commonality in the sensitized paths by many dynamic instances of a given static PC using the following expression. If ϕ is the set of gates in a circuit that change state in every dynamic instance of a static PC, and ψ is the set of gates that change state in at least one dynamic instance of the same static PC, then the commonality in sensitized gates for that PC is calculated using the ratio $\frac{\phi}{\psi}$.

S1.2.1 Core Microarchitecture

Using the Fabscalar Core-1, we generate synthesizable RTL design of a 4-wide out-of-order microprocessor core with 32 entry instruction queue, 96 entry physical register file and fetch-to-execute pipeline depth of 10 (Figure 1).

S1.2.2 Core Components

The following four microprocessor components are selected for this study. Together, they cover a wide spectrum of micro-architectural events for studying instruction level commonality in sensitized paths.

- 32-bit *Simple ALU*: We select this component as it contains a high logic depth compared to most other structures in a microprocessor core. Consequently, the ALU provides an interesting structure to study commonality in sensitized paths. It also offers a way to compare our results with the existing work.
- *Issue Queue Select*: This unit implements the instruction selection logic in the processor. Given a request vector from the existing instructions in the issue queue, it picks up to four instructions to be scheduled. The selection logic sets the request grant line for the selected instructions. Due to frequently repeated patterns in instruction selection, we expect a high degree of commonality in this structural component.
- *Address Generation Unit* (AGEN): This module represents the effective address computation necessary during typical load and store instructions. A given static instruction can compute different addresses in its various instances. However, often these effective addresses differ by a single bit (e.g., while looping through an array structure), resulting in excellent similarity in the sensitized logic paths.
- *Forward Check Logic*: This unit controls the latches in the bypass network to ensure correct execution of back-to-back de-

Scheme	Overhead (Scheduler only)			Overhead (core-level)		
	Area	Dynamic Power	Leakage Power	Area	Dynamic Power	Leakage Power
ABS	0.77%	0.57%	0.87%	0.03%	0.05%	0.01%
FFS	0.77%	0.57%	0.87%	0.03%	0.05%	0.01%
CDS	6.35%	1.56%	6.80%	0.24%	0.13%	0.08%

Table 2: Area and Power overhead of proposed VTE.

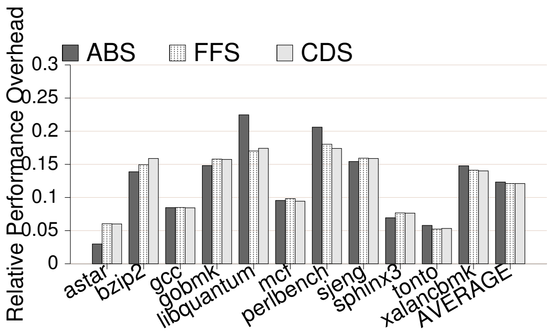


Figure 8: Performance Overhead Comparison During Faulty Execution at high fault rate (0.97V), normalized to EP [12,13]. (Lower is better.)

pendent instructions. If dependency conditions are met, then the output from a functional unit is latched directly to one of the inputs in the same or another functional unit. From a given instruction perspective, these logic steps will compute identical latching of outputs as long as scheduling decisions remain identical. Since the code path followed in a program often recur, instructions behind a given instruction tend to recur frequently, leading to identical scheduling decisions. Thus, we expect a high degree of similarity in the sensitized paths.

These components are synthesized using the Synopsys Design Compiler tool. Table 3 presents the characteristics of the synthesized processor components. The size and complexity of these structures vary substantially. For example, *Simple ALU* has the largest size (4728 gates) and greatest logic depth. In comparison, the *Forward Check* module is substantially smaller. Together, these structures represent a range of sizes and complexities of sub-modules expected in modern pipelined microprocessors.

S1.3 Results

Figure 7 presents the commonality in sensitized paths for several benchmark programs in the four selected components. The results show the weighted average, based on frequencies of each instruction, of all dynamic instances from the static instructions exercising the units. We see a substantially high commonality in the paths sensitized across a wide range of real programs. On an average, we observe a 87.4%, 89%, 92.4% and 90% commonality in the issue queue select, address generator, forward check logic and ALU, respectively. Certain benchmarks like *vortex* show an extremely high commonality (e.g., 96% in the issue queue) as it operates on a smaller range of input values.

S1.4 Opportunity For Predictive Scheduling

The collective high commonality in the sub-modules indicates a high timing violation predictability from instruction PCs. This predictability opens up a whole new class of violation mitigation techniques, radically marginalizing the overhead of tolerating timing violations. Using precise information about a timing violation, several clock cycles ahead, it is possible to design violation aware scheduling techniques as presented in this work, to tolerate the violation with a minimum performance overhead.

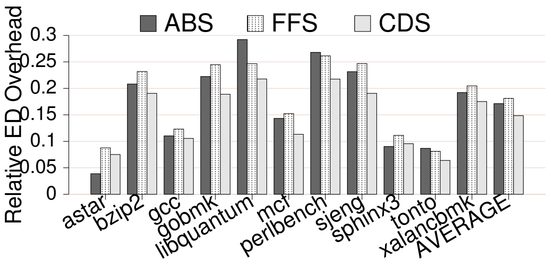


Figure 9: ED Overhead Comparison During Faulty Execution at high fault rate (0.97V), normalized to EP [12, 13]. (Lower is better.)

S2. HIGH FAULT RATE ENVIRONMENT

During a higher fault rate (V_{DD} = 0.97V), our scheme reduces the performance overhead by 88% on an average across various benchmarks (Figure 8). In certain benchmarks like *libquantum*, both FFS and CDS are more effective compared to ABS, achieving 83% overhead reduction compared to 78% reduction. On the other hand, in *astar*, ABS outperforms both FFS and CDS. On an average, our schemes reduce the ED overhead by 83% (Figure 9).

S3. AREA AND POWER OVERHEAD

Table 2 presents the overhead of our proposed schemes compared to the scheduler in the baseline machine that tolerates timing violations through error padding. Our proposed schemes ABS and FFS utilize the same fundamental logic in scheduling and tracking functional unit status, while CDS needs additional logic to dynamically assess the criticality of instructions. At the entire core level the scheduler consumes 3.9% area, 8.9% dynamic power, and 1.2% leakage power. Thus, the overheads at the entire core level are minimal in our schemes (e.g., 0.24% area overhead, 0.13% dynamic power overhead, and 0.08% leakage power overhead in CDS).