Predicting Timing Violations Through Instruction-Level Path Sensitization Analysis

Sanghamitra Roy  Koushik Chakraborty

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

ABSTRACT
In this paper, we present a novel technique for early prediction of timing violations in high-performance pipelined microprocessors. We show that a static instruction in a microprocessor, identified by its Program Counter (PC), is an excellent predictor of an upcoming timing violation. Our analysis combines architectural data collected from real program execution with gate level logic analysis. Exploiting this unique property, we propose a robust system design that predicts and tolerates timing violations seamlessly in a pipelined microprocessor. Under two different faulty environments, we show 20.9–89.8% and 14.6–80.6% average performance improvements in real programs over other state-of-the-art techniques, respectively.

Categories and Subject Descriptors
B.8.1 [Hardware]: Reliability, Testing and Fault Tolerance

General Terms
Reliability

Keywords
Timing Faults, Path Sensitization

1. INTRODUCTION
Current and future technology nodes are increasingly susceptible to timing violations, an artifact of rapid technology scaling. Guided by a combined effect of static (process variation and wearout) and temporal (thermal, voltage or utilization) variation, timing violations can occur sporadically [14, 12]. Consequently, there is a growing need for runtime timing error detection and correction techniques in robust microprocessor designs.

A key factor behind these timing violations is the specific critical path sensitized by an instruction in a circuit block. In this paper, we demonstrate that an instruction in a microprocessor, identified by its Program Counter (PC), is an excellent predictor of an upcoming timing violation. This intriguing property—a novel contribution of our work—is due to the extensive commonality in critical paths sensitized by a given instruction during its repeated execution. Using this property, it is possible to get an indication of a timing violation in a later pipeline stage early: several clock cycles in advance.

This early indication of an upcoming timing violation can enable a host of robust system design mechanisms, otherwise impossible with existing state-of-the-art techniques. Popular error detecting techniques like RAZOR and the widely pro-posed embedded timing sensors are unable to predict timing violations several clock cycles in advance [6, 7]. Our work is the first of its kind that demonstrates the instruction level predictability of timing violations in pipelined microarchitectures.

Exploiting this timing violation predictability, we design a robust system that predicts and tolerates timing violations seamlessly in a pipelined microprocessor. We use a low overhead instruction level Timing Violation Predictor (TVP) that predicts upcoming timing violations in as early as the decode stage. Using the early prediction capability, we can then enable a pipeline stall signal at the appropriate cycle. Our proposed design can effectively handle timing violations in complex microprocessor designs with forwarding logic and back-to-back dependent instruction scheduling.

Existing works on timing violation detection and correction techniques fall under two broad categories: reactive and proactive. Reactive techniques detect a fault after its occurrence [6, 2]. Subsequently, the fault can be corrected in a pipelined processor by using an instruction replay, which results in a large performance overhead [5]. Proactive techniques, on the other hand, detect an upcoming violation before the clock edge using various sensors embedded in the pipe stages [3, 7]. Once detected, many such techniques use time borrowing to tolerate the violation, avoiding any performance impact. In this paper, we show that time borrowing fails to suffice as a general robust technique in high-performance microprocessors. This is because time borrowing cannot be used in forwarding bypass networks that are an integral part of microprocessors for speedy execution of back-to-back dependent instructions.

We make the following contributions in this paper:

• We show that a given instruction sensitizes similar logic paths in various circuit blocks during its repeated execution (Section 2). Our rigorous analysis integrates architectural simulation data with a gate level logic analyzer to determine critical paths sensitized during a program execution.
• Exploiting this unique property, we propose a timing violation predictor (TVP) design and its usage in a mod-
ern microprocessor (Section 3).
- Using a circuit-architectural analysis, we illustrate limitations of time borrowing techniques in high-performance microprocessors (Section 4).
- Using a full-system architectural simulation, we demonstrate average performance improvements of 20.9–89.8% and 14.6–80.6% over other state-of-the-art techniques under two diverse faulty environments, respectively (Section 6).

2. PATH SENSITIZATION: AN INSTRUCTION LEVEL PERSPECTIVE

In this section, we demonstrate the strong correlation between a given static instruction and the critical paths sensitized by many of its dynamic instances in a pipe stage (Section 2.1). This strong correlation results in the predictability of timing violations from the static instruction. This fundamental observation is derived through a rigorous circuit-architectural analysis (Section 2.4). We combine data collected using architectural simulation (e.g., instruction type and input) with a logic analyzer to identify specific sensitized paths, primary inputs, and existing logic state of the gates in the micro-architecture component.

2.1 Instruction and its Sensitized Paths: Correlations

During the program execution, each instruction has many dynamic instances. These dynamic instances show a striking commonality in their sensitized paths through a given pipeline stage. Such high commonality stems from several key factors: (a) identical changes in the micro-architecture state; (b) stability in instruction inputs; (c) identical nearby instructions during its repeated execution ensuring similar internal logic state prior to each execution. Some examples of this commonality can be traced to a few well known characteristics of instruction execution (details in Section S1):

- For a certain PC, the decode stage interprets identical instruction bits in the same way each time.
- An instruction causes identical wakeup select activity in the Scheduler in all its dynamic instances.
- A small set of load instructions cause repeated cache misses sensitizing similar logic paths.
- Stable input values of ALU instructions [17].

2.2 Case Study with ALU

We now present a case study with ALU instructions. The ALU presents an intriguing case study for this analysis as it contains some of the deepest logic paths among all the structural units in a processor. Consequently, the likelihood of diversity in the sensitized paths through an ALU is higher than most structures in a microprocessor. We discuss other sources of commonality in details in Section S1.

For an ALU instruction, the existing logic state depends on the immediately preceding instruction that executed on that ALU. Since instructions are drawn from a static code, it is expected that the neighboring instructions are fairly stable (barring occasional wrong path executions where instructions enter the pipeline due to a mis-predicted branch). On other hand, the ALU inputs supplied by the instruction are determined by the specific task accomplished by the instruction. For example, several instructions have one of the operands encoded with the instruction, thereby stabilizing one of the input values in all their dynamic instances.

Branch History: During the program execution, the code path followed before the execution of a particular instruction plays a critical role in determining the specific inputs for that instruction, as well as, the preceding instruction scheduled on the ALU. Hence, we also inspect the commonality seen for a combination of the recently followed code path and the static PC. In our analysis, we use outcomes of the last 32 branches to encode the recently exercised code path.

2.3 Dynamic Path Sensitization of an Instruction

Let us consider an instruction, "add R1, R2, R3", where the sum of register contents in R1 and R2 is stored in R3. This instruction may execute twice (dynamic instances), but with different input values in R1 and R2. These dynamic instances may then sensitize specific paths in a circuit component. Our goal is to find the commonality among all such sensitized paths from a given instruction.

Figure 1 illustrates this concept of dynamic path sensitization of a particular instruction denoted by \( I \) in a circuit with seven nodes. Two dynamic instances of \( I \), \( I_1 \) and \( I_2 \), sensitize the paths shown by the shaded areas. \( S_c(I_1) \) indicates the set of nodes that change state while executing instruction \( I_1 \). Hence, we see that \( S_c(I_1) = \{1, 2, 5, 7\} \) and \( S_c(I_2) = \{2, 3, 5, 6, 7\} \).

The set of common gates sensitized by \( I_1 \) and \( I_2 \) is given by the intersection \( S_c(I_1) \cap S_c(I_2) = \{2, 5, 7\} \). The union of the two sets is given by \( S_c(I_1) \cup S_c(I_2) = \{1, 2, 3, 5, 6, 7\} \). This analysis shows that the two dynamic instances of \( I \) have a 50% commonality in sensitized paths, calculated using the ratio \( \frac{|S_c(I_1) \cap S_c(I_2)|}{|S_c(I_1) \cup S_c(I_2)|} \).

2.4 Methodology

Performing logic analysis using architectural data presents both computational and methodological challenges. A full program run consists of trillions of instructions [8]: analyzing the input and the corresponding sensitized paths from each instruction is computationally prohibitive. Even representative phases of the program consists of hundreds of million instructions, posing a massive computational challenge. To tackle this challenge, we adopted several important steps. First, we use representative phases of 100 million instructions of several SPEC CPU2006 benchmarks using the SimPoint toolset [15]. Second, we use a profiling run of architectural simulation to identify the top 100 instructions that most
Figure 2: Estimating the commonality in sensitized paths. The dynamic instances of an instruction $I$ and the corresponding inputs $<I_i, in_j>$ are generated from the architectural simulation. Our in-house logic analyzer runs each instruction on the synthesized netlist and generates the set $S = \{I_i\}$ for each dynamic instance of $I$. The post-process step estimates the commonality in the sensitized paths of $I$ as shown in Section 2.3.

Figure 3: Commonality in sensitized paths (64-bit integer ALU from OpenSPARC T1).

frequently exercise the 64-bit ALU from these phases. Many of these benchmarks have over a million static instructions: keeping track of all of them is impractical. Third, we repeat the architectural simulation to collect inputs from all the dynamic instances of these top instructions. We also collect inputs and instruction type scheduled on the ALU right before a dynamic instance of these instructions. This is a necessary step, as we use preceding instructions to set the existing internal logic state of the ALU. Subsequently, when an instance of a top PC is analyzed, we can identify the sensitized path. Figure 2 shows an overview of our methodology.

2.5 Results

Figure 3 shows the commonality in sensitized paths on a 64-bit integer ALU from the OpenSPARC T1 processor. The result shows the weighted average, based on frequencies of each instruction, of all dynamic instances from the top 100 static instructions exercising the ALU.

We notice a substantial commonality in the sensitized paths across a wide range of applications. On an average, we observe 79.7% commonality with just the static PC. When the branch history is combined, the commonality increases substantially (average of 87%). Certain benchmarks show tremendous commonality. For example, gcc and perlbench show 96.7% and 94.6% commonality, respectively. Instructions from these benchmarks operate on a fairly small range of input values, leading to this result. In contrast, hammer shows relatively poor commonality as it has substantial data diversity.

2.6 Significance

The large commonality in sensitized paths from an instruction indicates that if one of its instances cause a timing violation, subsequent instances are highly likely to cause violations under identical operating conditions (voltage, temperature). Thus, we can predict these timing violations ahead in time, using a combination of instruction PC and the recent branch history.

Such early prediction of timing violations presents a tremendous opportunity for designing robust pipelined systems. Unlike Razor, where timing violations are detected only after they have actually taken place [6], precise information about an upcoming timing violation is available several clock cycles before. Consequently, it is possible to set up stall signals in appropriate pipe stages through this technique. Razors is unable to do so in aggressively clocked microprocessors due to insufficient time in propagating the stall signal throughout the requisite pipe stages [6]. Based on this unique opportunity, we now propose a novel system design.

3. SYSTEM DESIGN USING TIMING VIOLATION PREDICTOR

In this section, we discuss our proposed techniques for designing a robust pipelined architecture that exploits the prediction of upcoming timing violations (Section 3.1). Using a Timing Violation Predictor (TVP) (Section 3.2), we can enable the pipeline stall signals to tolerate timing violations and preserve correct execution.

3.1 System Overview

We need a single TVP for the entire pipelined microprocessor. This TVP predicts the occurrence of timing violations in the various pipeline stages for different instruction PCs. During the decode stage, each instruction is checked for a possible timing violation using the TVP.

There are two possible outcomes from the TVP for any instruction:

- **Predicted No-Violation**: No timing violation is predicted by the TVP. Instructions proceed normally.
- **Predicted Violation**: A timing violation is predicted to occur at a specific pipe stage. A stall signal is then initiated at the appropriate stage of the pipeline. For example, if an instruction is decoded with a predicted violation at stage 5, then the stall signal is enabled in the pipeline when that instruction enters stage 5. This stall signal allows the pipe stage 5 to complete in two clock cycles, while the input to all other stages are recirculated to avoid forward flow of instructions during that cycle.

We use already existing circuitry in high performance microprocessors to implement pipeline stalls, and require minimal modification to simply enable the stall signal when necessary. The stalls are necessary in a pipelined architecture for a variety of reasons. One of the main reasons behind these stalls is correctly tackling data dependency, where an instruction cannot proceed due to the unavailability of input com-
puted by an older instruction (e.g., an instruction depends on a load instruction, but the load may miss in the cache causing several cycles delay) [13].

The TVP cannot rectify timing violations in the fetch and decode stages, so these are mitigated through our recovery mechanism (discussed later). However, violations in these early pipe stages are rare, as temporal variations like thermal and voltage fluctuations are predominant in the back end of the pipeline [10].

**Handling Mis-prediction:** The predictor can mis-predict in two ways:

- **False Positive:** If the TVP predicts a timing violation, whereas one does not actually occur, we incur the loss of one clock cycle. However, we do not face any correctness problem in the pipeline.
- **False Negative:** If the TVP does not predict a timing violation, whereas one actually occurs, we may encounter an error. In this case, an error detection and recovery mechanism is fired that detects and corrects the timing violation using a pipeline flush.

**Error Recovery and Correction:** Each pipeline stage in our system is equipped with error detection and recovery circuitry. The error recovery is fired only when an error is detected and there is no scheduled stall signal (during a false negative TVP prediction). This situation essentially implies a timing violation where no corrective measures have been taken. To recover from this error, we initiate a pipeline flush, thereby avoiding the propagation of incorrect value in the system. We use the voltage glitch detector circuit proposed in [4] as our error detector. This circuit requires substantially lower power and area overhead compared to techniques using duplicated shadow flip-flops [6].

### 3.2 Timing Violation Predictor (TVP)

Fundamentally, timing violations in a pipe stage depend on: (a) the operating conditions (e.g., local temperature, voltage fluctuation), (b) combination of process variation and aging degradation, and (c) the sensitized path. We combine the history of timing violations from an instruction (Section 2.6), and on-chip thermal and voltage sensors to design the TVP.

The TVP consists of three major components: (a) combination of PC and branch history; (b) Violation History Table (VHT); and (c) thermal and voltage sensor. Section 2 demonstrated the strong correlation between timing violations in a pipe stage with a combination of an instruction and recent branch history. We exploit this program behavior by creating an index using both the PC and the Branch History Register (BHR), which records the last 32 branch outcomes from the program execution.

Each entry in the VHT maintains a combination of PC and BHR that caused a recent violation, associated tag from the PC, and the pipe stage where the violation occurred. The thermal and voltage sensors provide an indication if the current condition is likely to cause a timing violation. During decode, a timing violation is predicted when there is a successful tag match in the VHT, and the sensors indicate possible conditions for timing violations.

The VHT is updated after all mis-predictions. During a false positive, an existing entry is invalidated, making it available for new faulty instructions. During a false negative, a new entry is inserted in the table, which may lead to the eviction of an existing entry.

![Figure 4: Timing Violation Predictor (TVP) Design](image)

**3.2.1 Predictor Performance**

Superficially, it may appear that effective prediction requires a large VHT to avoid address collisions in the table. In reality, a small to moderately sized VHT is sufficient as it only tracks instructions causing timing violations.

For example, thermal emergencies in a chip may last a few milliseconds, depending upon its thermal constants [16]. During these intervals, a program may execute a few million dynamic instructions, but timing violations may be restricted to a small percentage of static instructions. We find that using a 4K sized VHT and 10% faulty dynamic instructions in such intervals, leads to less than 2% false negatives from address collisions.

### 4. RELATION WITH POPULAR TIMING VIOLATION TECHNIQUES

In this section, we discuss other state-of-the-art techniques to mitigate timing violations in pipelined architectures. After briefly describing Razor in Section 4.1, we focus on time borrowing techniques that were proposed to mitigate the large performance overhead from Razor’s error correction mechanism. Combining a circuit-architectural analysis, we demonstrate how time borrowing techniques can be ineffective in masking timing violations in pipelined high-performance microprocessors. This important design issue stems from the prevalence of back-to-back dependent instructions (BDI) during program execution in these chips.

#### 4.1 Razor

Razor, proposed by Ernst et al. [6], consists of detection and correction mechanisms for timing faults. The timing faults are detected by comparing combinational logic outputs captured at the regular clock and a delayed clock using shadow flip-flops. The correction mechanism, in a high-performance pipelined architecture, involves recovery using counter-flow pipeline. A subsequent work from the same research group have used instruction replay [5]. Even with an in-order machine, they report a large performance overhead: about 1% error rate results in nearly 10% performance overhead [5]. To mitigate this large overhead from Razor, several time borrowing techniques were subsequently proposed.

#### 4.2 Time Borrowing Techniques

Time borrowing techniques constitute a class of techniques that can mask a timing violation by borrowing slack time from the adjacent pipe stages [3, 7, 4]. The promise of time borrowing techniques lies in the fact that if successful, they cause zero performance overhead. However, such techniques strongly depend on the availability of slack in some of the pipe stages to mask the timing violations in other pipe stages.
4.3 Problem with Time Borrowing Techniques

Time borrowing can fail when two connected critical paths in adjacent pipe stages are sensitized. Connected critical paths have a single flip-flop acting as the sink node of the first stage as well as the source node of the second stage. Using a static analysis of critical paths, Choudhury et al. show that such connections are rarely sensitized, thereby estimating a high success rate of time borrowing techniques [3].

Using only static analysis, however, grossly underestimates the actual sensitization of connected critical paths during program execution. Static analysis can account for all possible critical path connections, but it cannot determine the utilization rate for various connections at runtime. The frequent occurrences of the BDI issue in high-performance microprocessors during real program execution, massively increases the sensitization of connected critical paths. We now present data showing the prevalence of BDIs, and a detailed analysis of the design complexity in handling them.

4.3.1 Back-to-back Dependent Instructions (BDI)

Modern high-performance microprocessors heavily depend on the ability to issue BDIs to reap performance benefits. Using full system simulation (details in Section S3), we observe that 9–36% of instructions constitute BDIs in a high-performance microprocessor—a substantial portion. Similar results were also seen by other works [11].

4.3.2 Complexity of Handling BDIs

Figure 5 shows a bypass path in a typical pipelined microprocessor. The output of the arithmetic and logic unit (ALU) is latched in the EX/MEM register and forwards back to the ALU via the combinational forwarding unit and a combinational multiplexer (MUX) as shown in the figure. This bypass network helps in forwarding the ALU output to a back-to-back dependent ALU instruction in the immediate next clock cycle. Such forwarding logic, commonplace in high-performance microprocessors creates a cyclic pipeline with a single sequential register. Figure 5 also highlights a critical path in this circuit, where a flip-flop in the EX/MEM register constitutes both the source and the sink.

Now let us assume that the ALU has a timing violation in clock cycle \( t \), and is followed by a BDI in clock cycle \( (t + 1) \). Since the BDI uses the cyclic pipeline, cycle \( t \) needs to borrow slack time from the next cycle of its own block. At cycle \( (t + 1) \), there can be another timing violation in the dependent instruction. This is because the primary forces behind a timing violation are the degree of wearout of the circuit combined with the localized temporal variation (temperature and voltage conditions). Such localized temporal variation implies repeated timing violations in a specific pipeline stage/component for several cycles [1]. In such a case, cycle \( t \) is unable to borrow time from its own stage in the next clock cycle \( (t + 1) \), making it impossible for time borrowing techniques to rectify the timing violation.

5. ARCHITECTURAL METHODOLOGY

In this section, we describe our architectural simulation methodology for performance tradeoff analysis between various techniques to mitigate timing violations.

Core Microarchitecture: We use full-system simulation built on top of WindRiver SIMICS [9]. SIMICS provides the functional model of several popular ISAs, in sufficient detail to boot an unmodified operating system. For our experiments, we use the SPARC V9 ISA, and use our own detailed timing model to enforce the timing characteristics of a 4-wide out-of-order microprocessor. The core microarchitecture has 12 stages, 64 entry instruction window, 96 entry physical register file, and uses the YAGS predictor with 16K predictor table. Our TVP uses 4KB VHT, and shares the BHR with the YAGS. 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.

Workloads: We use several SPEC CPU2006 benchmarks, and focus our architectural simulation on representative phases extracted using the SimPoint toolset [15].

5.1 Fault Simulation Methodology

The goal of our fault simulation methodology is to analyze the performance during faulty execution. Among temporal variation, temperature induced faults may last for several milliseconds, while voltage fluctuation lasts for much smaller time frames. Consequently, we randomly select several 10ms time windows from the SimPoint phases in our benchmarks, and inject faults during these time windows. To cover a range of faulty executions, we show results when 1% of instructions experience faults, as well as, when 10% of instructions experience faults. All simulations are run with warmed up caches, so that cold cache effects do no mask other important performance artifacts.

Modeling Sensitized Path Commonality: When an instruction incurs a fault in a specific pipe stage, subsequent dynamic instances of the same instruction have a high probability of incurring faults (Section 2) in the same pipe stage. To model this behavior, we assign a fault probability to every faulty static instruction. For each benchmark, we set this probability to the percentage commonality from Figure 3.

6. EXPERIMENTAL RESULTS

In this section, we present the performance analysis of our technique compared to state-of-the-art techniques.

6.1 Comparative Schemes

- Razor: We model the Razor scheme for detection and correction of timing violations. The correction mechanism uses instruction replay, based on their most recent work [5].
- Time Borrowing: We have shown that time borrowing can fail to rectify timing violation errors in high-
6.2 Performance Impact

Performance loss from a pipeline flush or a stall greatly depends on the intrinsic characteristics of a benchmark. Typically, benchmarks with high instructions per cycle (IPC) are more affected by these corrective measures. Timing of these events is also critical. For example, when the instruction window is full due to a pending load miss, a pipeline stall and flush have marginal and large overheads, respectively.

Figures 6 and 7 show the performance impact of various schemes we compare when 1% and 10% of all instructions incur timing violations, respectively. The performance losses are shown relative to a fault-free baseline. We notice that hmmer suffers a large performance loss, especially for TBH. In hmmer, the out-of-order microarchitecture is able extract substantial instruction level parallelism due to the low percentage of dependent instructions (Figure 8). Thus, even with the lower recurrence of BDIs, time borrowing suffers substantial performance loss in hmmer, as an instruction replay has large overhead in hmmer than in lower IPC mcf or xalancbmk.

Despite using optimistic models for time borrowing techniques in an out-of-order core, our proposed scheme shows substantially better performance in a faulty environment. For 1% faults, our TVP outperforms time borrowing techniques by 74.1%, 60.6%, and 20.9%, respectively. For 10% faults, we notice average performance improvements of 72.2%, 54.9%, and 14.6%, respectively. Razor performs worse than time borrowing as it requires instruction replay for all faulty instructions.

7. CONCLUSION

We present a novel technique for early prediction of upcoming timing violations in a pipelined microprocessor using a combination of the instruction PC and temporal sensors. We exploit this phenomenon to design a robust system that predicts and tolerates timing violations seamlessly in a pipelined microprocessor.

Acknowledgments

This work was supported in part by National Science Foundation grant CNS-1117425.

8. REFERENCES

Supplemental Materials

S1. ARCHITECTURAL EVENTS LEADING TO COMMONALITY

Across many dynamic instances, a given instruction causes similar state changes in various pipe stages. In this section, we present a detailed discussion of many such changes in different stages of a pipeline. All of the following factors influence the high commonality in the sensitized paths of a instruction PC in its many dynamic instances.

- **Decode**: The bits encoding a given instruction A remains unchanged every time that instruction is decoded. Consequently, these bits are interpreted in an identical fashion every time, producing the same logic output from the decode stage. Moreover, the internal logic state of the combinational logic in the decode stage is set by the instruction immediately preceding the instruction A. Except for branch instruction and its target, the order of instructions remains fixed. Even for branch and target instructions, programs tend to show good predictability: leading to the low mis-prediction in modern branch predictors. These factors collectively combine to sensitize near identical logic paths in the decode stage for the instruction A, in all of its dynamic instances.

- **Rename**: In the rename stage, decoded instructions are allocated free physical registers to store their respective results. In addition, for any instruction, the rename logic determines the previously renamed instructions that the given instruction depends on. Since this dependency of an instruction remains unchanged during all of its dynamic instances (a fundamental property of the static code), the computation to accomplish this follows a similar path, sensitizing identical logic gates.

- **Schedule**: A substantial portion of the state change triggered during the issue of an instruction is directly derived from the instruction itself. For example, the type of instruction (e.g., load/store or ALU operation) determines the functional unit where the instruction is scheduled. Thus, resizing the identical functional unit in all of its dynamic instances sensitize the same path for a given instruction. Other important activities like wakeup and select are also heavily correlated to the same instruction PC. The wakeup logic is invoked when an instruction produces a new value in its destination register (through a load completion or ALU), marking its dependent instructions operand ready field [11]. Since the number of dependent instructions behind a given instruction depends on the static code, and thus remains stable, every dynamic instance of the given instruction causes highly similar state changes in the wakeup/select logic of the scheduler.

- **Memory Access**: Instructions show a markedly predictable pattern in their execution characteristics during the memory access stage. For example, a large number of cache misses result from a few select instruction PCs [SR2, SR3]. A cache miss and a cache hit sensitize orthogonal set of logic paths in the circuit responsible for memory access. Therefore, we expect a very high commonality in the logic paths in this stage as well. Other key components like load-store queues in a processor also observe predictable and repeated state changes from different load and store instructions. For example, past research has demonstrated that only a few select store instructions conflict with younger loads, thereby preventing them to be issued to the cache memory early [SR1]. Thus, actions necessary to enforce this dependence at runtime requires distinct state changes (but predictable through store PCs) from the default case. These factors again combine to produce a striking commonality in the sensitized logic paths in the memory access phase.

S2. PREDICTOR OVERHEAD

To estimate the power overhead of our Timing Violation Predictor (TVP), we design a TVP in Verilog using Figure 4. Our TVP uses a 4KB Violation History Table (VHT). The instruction PC and BHR which are accessed by the TVP are already a part of the microprocessor, and hence not included inside our TVP module. We next synthesize our TVP using the Synopsys Design Compiler and a 45 nm TSMC standard cell library and measure the dynamic and leakage power. To get a relative estimate of power overhead, we also synthesize a 96 entry Physical Register File using identical methodology and measure its dynamic and leakage power. We find that the TVP consumes 15.42% power relative to the register file. Most of the power in small SRAM structures (e.g., the predictor or the register file) with low associativity (direct mapped) goes towards driving the address and data bits. Compared to the register file that drives a 64-bit data, the predictor only drives 5-bit data (pipe stage encoding). Consequently, the predictor has substantially lower power overhead compared to the register file. Assuming that the register file consumes 6.6% of the core power, a reasonable estimate in a high-performance out-of-order core [SR4], the TVP consumes only 1.02% of the core power.

S3. PROBLEM WITH TIME BORROWING
To elaborate on Section 4.3.1, we perform a detailed analysis using full system simulation of a typical 4-wide out-of-order microprocessor core to estimate the percentage of back-to-back dependent instructions (BDIs). Figure 8 shows the percentage of instructions issued that depend on the result computed in the immediately preceding cycle. We observe that 9–36% of instructions constitute BDIs in a high-performance microprocessor—a substantial portion.

S4. REFERENCES


