Strategies for Optimal Operating Point Selection in Timing Speculative Processors

Omid Assare and Rajesh Gupta
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093–0404
{omid, gupta}@ucsd.edu

***Abstract*—Performance of timing speculative processors relies on strategies for accurate prediction of optimal operating points. In this paper, we develop an efficient process-variation-aware simulation framework and use it to evaluate a range of such timing speculation strategies. Our experiments on a timing speculative processor running applications from the MiBench benchmark suite show that, in a typical case, while a perfect timing speculation strategy can improve throughput by up to 143% over a guardbanded design, the most commonly used approach in the literature achieves only a 21.8% of the potential gains. By improving the speculation accuracy, the new strategies we propose in this paper can realize up to 35.6% of the potential gains, a throughput improvement of 50.9% over a guardbanded design.**

I. Introduction

Traditional sequential circuit designs work under the strict condition that voltage and frequency of the circuit should be set such that no timing violations can occur. An increasingly popular alternative approach called timing speculation [1], on the other hand, allows timing errors by removing timing margins and relies on circuit- and microarchitecture-level techniques to detect and recover from these errors and guarantee correct execution. The variability in how different instruction sequences cause errors leads to increased performance during the execution of less vulnerable code, while faulty instructions are detected and corrected using special hardware and by paying a recovery penalty.

Our work focuses on timing speculative processors that use frequency to tune their operating point, but our method is orthogonal to dynamic voltage scaling. Performance of these processors is determined by two competing mechanisms. While increasing the frequency improves processor throughput by fitting more clock cycles into a fixed amount of time, it also increases the rate of timing errors because more paths fail timing requirements. The goal of dynamic frequency tuning is finding the frequency that balances these effects such that processor throughput is maximized. Accordingly, timing speculative processors track the error rate during the execution and use it as a feedback mechanism for tuning their frequency. For instance, the conventional approach used in most related proposals periodically samples the processor error rate and tries to keep the long-term error rate close to a pre-specified threshold by increasing (decreasing) the frequency when the error rate is below (above) the threshold. In this paper, we examine a range of strategies that a timing speculative processor can adopt to dynamically select the optimal operating point.

We also tackle the problem of simulating these systems which is necessary to gain insight into their timing behavior as well as comprehensive design space exploration. Error rate models typically used for analyzing timing speculative processors do not get more sophisticated than assigning different numbers to various hardware/software components and/or operating conditions, and the role of software in sensitizing timing paths and changing error rates is often ignored. It is even common to use the simplest possible model, a fixed number, for error rate of the system at all times [2], [3], [4]. In the absence of efficient error models and when more accuracy is needed, researchers are left with no choice other than using time-consuming low-level timing simulations, forcing them to limit the analysis in both time (analyzing only small parts of the application software) and space (analyzing only a few components of the system), adversely affecting optimization opportunities and/or accuracy of evaluation [5], [6], [7], [8].

In this paper, we develop an efficient instruction level error model and use it in to create a fast process-variation-aware simulation framework suitable for the study of timing speculation.

Contributions of this work have been summarized below:

1) We introduce and analyze three timing speculation strategies. First, we argue that frequency tuning should be directed by software and performed at the basic block level where instruction sequence is fixed and predictions are likely to be more accurate. Second, we show that error rate sampling should be temporally limited because the most recent history of errors is often a better predictor of timing behavior. Third, we propose a more robust scheme for dynamic frequency tuning by relying on an optimization algorithm instead of threshold-based control. Finally, we describe the design of a new timing speculation scheme based on these strategies.
2) We develop a simulation framework for evaluating the performance of timing speculative processors. The framework creates an instrumented version of the program that simultaneously implements (i) a process-variation-aware instruction level error model to predict timing errors and estimate error rates as well as (ii) the dynamic frequency

tuning mechanism necessary to realize potential gains of timing speculation. This method results in very high speed simulations because instead of using a micro-architecture level simulator, they are performed by running the instrumented program on any machine that implements ISA of the target processor.

3) Using our simulation framework, we tune the design parameters of our timing speculation scheme and evaluate its performance. We show that our method improves processor throughput by more than 50% over a conventional guardbanded design while incurring little power overhead. To put this improvement in perspective, we evaluate the potential gains of an ideal timing speculation scheme that can perfectly track the timing behavior of the system as well as the most popular method in the literature. We find that while the conventional approach can only realize around a fifth of the potential gains of timing speculation, even our efficient method leaves almost two-thirds of potential gains untapped.

II. Speculation Strategies

In this section, we describe the design of our timing speculation scheme while analyzing the three main speculation strategies it adopts.

A. Selective Local Speculation

A number of recent works have documented the concept of spatial *timing error locality* where static instructions exhibit consistent error behavior over a period of program execution [8]. To take advantage of this phenomenon, in *selective local speculation*, decisions for changing the frequency are made separately for some basic blocks. This is in contrast to the conventional approach where the processor only tracks and controls the global error rate. We expect that a speculation strategy that works on the basic block level where the instruction sequence is fixed can make more accurate predictions.

This scheme can be realized by maintaining a table of basic block error rates and frequencies tagged by the PC address of the first instruction in the basic block. We refer to this table as *timing speculation table*. At the basic block entry when PC points to the first instruction, the frequency is set to the value previously stored for the basic block. This value should then be updated with a new frequency prediction for the next execution at the basic block exit. To track when execution is exiting the basic block, the table also includes a counter initiated with the number of basic block instructions, N_I, at the entry of the basic block. This counter is decremented each time an instruction finishes execution, reaching zero at the basic block exit.

The additional costs over the conventional approach include the power and area of the timing speculation table as well as potentially more frequent frequency changes. To control these costs, we introduce two design parameters. The first parameter, N_B, limits the speculation instances spatially to the N_B most frequently executed basic blocks. These basic blocks are selected for local speculation because the accuracy of predicting their frequency affects running time more significantly than others. N_B determines the number of entries in timing speculation table. Similar to [7], we assume that the table is implemented as a SRAM structure and incurs a negligible power overhead as long as $N_B \leq 128$.

The second parameter, N_S, limits the speculation instances temporally by specifying how many times we skip prediction and reuse the previous frequency for a basic block before a new prediction is made. For example, $N_S = 4$ means that a predicted frequency will be reused for the next 4 executions of the basic block. This effectively limits the number of frequency changes for single basic block loops because the predicted frequency does not change for the next N_S iterations/executions. Similar to N_I, this can be implemented with a counter decremented with each basic block execution. New predictions are kept from updating the frequency field of the timing speculation table unless the counter value is 0. We will explore the design space created by these parameters in Section V.

B. Limited Error Sampling

Selective local speculation is based on the assumption that error rate of previous executions of a basic block is a better predictor of its future error rate than the global error rate. This local strategy raises the question of the appropriate depth of error rate sampling. To predict the future error rate of a basic block, should we implement a long-term sampling scheme using error rates of all previous executions of the basic block, or rely only on its most recent history and use, for instance, the only last n executions?

To answer this question, we designed and performed a simple experiment. In this experiment, we explored how the *effective delay* of an instruction defined as the propagation delay of the slowest path it sensitizes changes as it is executed multiple times. We selected the 5 most time consuming basic blocks of the programs in Mibench benchmark suite [9] and tracked the effective delays of their instructions as they were iteratively executed in loops. We then measured the *distance* between effective delays of dynamic instances of each instruction. Fig. 1 shows the average distance of the instruction effective delays as a function of their execution distance. Execution distance of two dynamic instances of an instruction executed in the kth and jth iterations of the loop is defined to be $|k - j|$. For example, the execution distance between two dynamic instances of an instruction executed in the first and second iterations of the loop is 1 while the first and third instances have an execution distance of 2. Instruction delay distances were measured as the Hellinger distance between the distributions. As Fig. 1 shows, there is a generally direct relationship between delay and execution distances. This implies that the most recent execution of a basic block is likely a better predictor of its next execution in terms of timing errors. Accordingly, *limited error sampling* uses the most recent error rate of a basic block to predict its next frequency.

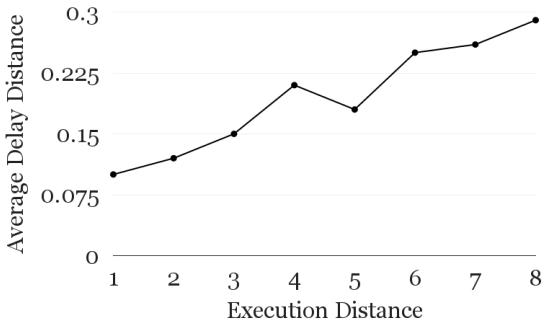


Fig. 1: Delay distance as a function of execution distance.

C. Maximum Throughput Tracking

In the conventional approach, frequency is adjusted so that the error rate remains close to a pre-specified error rate threshold. Consequently, the performance of this method is highly dependent on the selection of error rate threshold(s). In addition to the difficulty of finding optimal threshold values, this approach cannot capture the highly dynamic relationship of frequency and error rate. *Maximum throughput tracking* is a more robust strategy where error rate threshold is eliminated and frequency adjustments are made based on the dynamic changes of throughput rather than the raw error rate. Similar to the well-known hill climbing algorithm, the direction of frequency change in each iteration is determined based on the effect of the previous change on the throughput. Frequency is increased when a previous frequency increase (decrease) has resulted in an increase (decrease) in throughput. Conversely, frequency is decreased when a previous frequency increase (decrease) has led to a decrease (increase) in throughput. Similar to [7], we assume the hardware cost of implementing this algorithm is negligible.

III. Error Model

We use a functional timing model called *Clustered Timing Model* (CTM) [10] that enables dynamic timing analysis by grouping functionally similar timing paths of the processor and modeling their collective propagation delay as a function of their specific operation. Accuracy of CTM has been verified with a maximum error of 6.7% across a wide range of voltage-temperature corners [10]. Our approach in estimating effective delay of instructions is motivated by the observation that typical applications spend most of their runtime in loops, executing a few basic blocks over and over again. In order to take advantage of this, we develop timing models for each basic block in a pre-characterization phase where the most time consuming parts of the timing analysis are performed offline. In the next section, we will show how this approach allows the simulation to be performed at the architecture level, significantly improving the simulation time.

A. Clustered Timing Model

Overview. CTM partitions the endpoints of a digital circuit into a set of *Register Clusters (RC)* and the timing paths into a set of *hyperpaths* that connect the RCs together. We consider the integer unit of LEON3, an open-source in-order processor core that implements the SPARC V8 architecture [11]. As instructions go through the pipeline, they change the RC values. The model then includes a function for each hyperpath that predicts its effective delay (i.e. maximum propagation delay of its sensitized paths) based on its origin and destination RC value transitions. Finally, an instruction is predicted to cause a timing error when at least one of the hyperpaths it uses has an effective delay larger than the clock cycle.

Functional Paths. In order to map RC value transitions to hyperpath effective delays, CTM models each hyperpath, which is essentially a collection of timing paths, as a set of *functional paths*. The operation performed by a hyperpath is then viewed as a combination of the activation of some of its functional paths. As an example, consider the execution stage hyperpath when an `add` instruction is being performed. This hyperpath consists of the timing paths of the multi-bit adder in the execution stage. Roughly speaking, each bit in the output of the adder can go high using a carry chain that starts from a lower order position (we ignored the local activation when input carry is zero because carry chains are typically slower). Accordingly, CTM considers a functional path for every possible carry chain in the adder, from every bit position to all higher order ones. Functional paths of all hyperpaths are identified similarly based on their specific operation.

Training and Use. Training of a CTM involves characterizing the delay of functional paths. This is achieved by measuring the hyperpath delay when running special training codes designed to selectively activate specific functional paths. When process variation is considered, all delay values turn into random variables and the correlation between timing paths is abstracted into correlations between functional paths. To use the model, the delay of a hyperpath is calculated as the maximum of the delays of its activated functional paths rather than the activated timing paths.

Implementation. We implemented a CTM for LEON3 in a micro-architectural simulator, similar to the one described in [10], that takes a sequence of instructions and produces their effective delays. Since the model needs RC values at every clock cycle, the simulation cannot be performed at the architecture level and is, therefore, too slow for typical programs with large data sets. Throughout this paper, *measuring* instruction probabilities refers to using this CTM-enabled simulator to estimate them.

B. Control Delay Characterization

Distinguishing the data and control planes of the processor, our approach is based on the intuition that while the sensitized paths in the processor datapath vary each time a basic block is executed with a different input, control network paths go through similar activation patterns. Using CTM terminology, we propose to classify the hyperpaths into two types: (i)

control hyperpaths that together constitute the control network of the processor, and (ii) data hyperpaths that together form the datapath. Then, the effective delay of an instruction, D, is estimated as,

$$D = MAX\left(D_{control}, D_{data}\right), \tag{1}$$

where MAX represents a statistical maximum operation and $D_{control}$ and D_{data} are the effective delay of the control and data hyperpaths used by the instruction, respectively, hereafter referred to as the instruction control and data delays. Note that all delays are assumed to follow a Gaussian distribution.

By moving the estimation of control hyperpath delays to an offline pre-characterization phase, we expect to achieve significant simulation time improvements for the following two reasons. First, while data hyperpaths perform operations that can be concisely described mathematically, control hyperpaths have a more irregular functional path structure due to the bit-level computations they perform on control signals. More importantly, we can now perform the simulation at the architecture level because data hyperpath delays can be modeled using only architecturally visible registers whereas estimating control hyperpath delays requires values of internal pipeline registers, referred to as RCs in CTM terminology.

Therefore, in the pre-characterization phase, we measure the control delays of all instructions for each basic block. A complicating issue is the effect of the program control flow. Instructions of two neighboring basic blocks usually share the pipeline during their execution. As a result, control delay of an instruction could be different depending on the previous executed basic block. To account for this effect, we measure instruction control delays once for each possible previous basic block in the Control Flow Graph (CFG). Later during the simulation when the previous basic block is known, we use the appropriate control delay when evaluating Eq. 1.

Suppose that we want to characterize control delays of instructions in basic block B along one of its incoming edges e from basic block B'. We implemented a simple symbolic execution tool that derives the branch condition of B' in terms of its input (i.e. registers and/or memory locations accessed by instructions in B'). This condition is then used to ensure that the randomly generated input executes e. Finally, the execution is started at the top of B' and control delays of instructions in B are measured. This process is repeated multiple times and the mean of all measured control delays of each instruction is used during the simulation.

To evaluate the accuracy of our model, we randomly selected 100 basic blocks from the applications in MiBench [9] benchmark suite. These basic blocks contained an average of around 11 instructions. We characterized the basic blocks using 10 measurements for each control delay estimation using the method described above. Finally, using 100 randomly generated input vectors for each basic block, we compared the estimated and measured effective delays of all instructions. To quantify the comparison, we use squared Hellinger distance as a measure of the difference between instruction delay distributions. It takes values between 0 and 1 where smaller values indicate more accuracy. The squared Hellinger distance for two Gaussian distributions, $P \sim N(\mu_1, \sigma_1)$ and $Q \sim N(\mu_2, \sigma_2)$, is given by,

$$H^2(P,Q) = 1 - \sqrt{\frac{2\sigma_1\sigma_2}{\sigma_1^2+\sigma_2^2}}e^{-\frac{1}{4}\frac{(\mu_1-\mu_2)^2}{\sigma_1^2+\sigma_2^2}}. \tag{2}$$

We found that the distance was smaller than 0.1 in 97.3% of the experiments with an average value of 0.027, illustrating the reliability of the model.

C. Error Rate Estimation

During the simulation when frequency is known, instruction delays estimated by the error model must be converted into basic block error rates so that the simulator can implement frequency tuning. Since instruction delays are estimated as Gaussian distributions, the probability of an instruction experiencing a timing error, referred to as its *error probability*, is given by,

$$P = \frac{1}{2}\left[1 + \text{erf}\left(\frac{\frac{1}{F}-\mu}{\sqrt{2\sigma^2}}\right)\right], \tag{3}$$

where F is the working frequency, μ and σ are the mean and standard deviation of the instruction delay, and erf$(\cdot)$ is the error function.

Now, consider a basic block containing n instructions with error probabilities $p_1, \ldots, p_n$. To estimate the error rate, let $I = (I_1, \ldots, I_n)$ be a set of Bernoulli random variables corresponding to the instructions such that $Pr(I_i = 1) = p_i$. Clearly, the number of errors can be expressed as the sum of instruction random variables, $n_e = \sum_{i=1}^{n} I_i$. Therefore, the expected number of errors can be calculated by summing instruction error probabilities and the expected error rate is given by,

$$r_e = \frac{1}{n}\sum_{i=1}^{n} p_i. \tag{4}$$

IV. Simulation Framework

In this section we describe a framework for evaluation of various timing speculation strategies. The framework creates an instrumented version of the program that simultaneously implements the error model described in Section III to predict timing errors and estimate error rates as well as the timing speculation strategy described in Section II to perform dynamic frequency tuning. To perform the simulation, the instrumented program can be run on any machine that implements the instruction set architecture (ISA), resulting in very fast simulations. The operation of the instrumented program can be summarized in the following steps:

Before executing a basic block, error rate is read, frequency is set and pre-characterized instruction control delays corresponding to the executed incoming edge are loaded. **During** the execution, *transition signatures* of instructions are extracted and saved. Transition signatures, which will be explained shortly, are used to identify the functional paths

activated by an instruction. **After** executing the basic block, activated functional paths are identified from transition signatures and used to estimate data delay of instructions. Then, effective delays of instructions are estimated using data and control delays and are used to find error probabilities of instructions using current frequency. Finally, the expected error rate of the basic block is computed, recorded and used to calculate speculation speedup.

A. Transition Signatures

Suppose that we are interested in finding the functional paths activated by the current instruction, i^c, with operands op_1^c and op_2^c and result r^c, which is executed immediately after the previous instruction, i^p, with operands op_1^p and op_2^p and result r^p. Below, we define a set of *transition signatures* derived from these architecturally visible parameters that uniquely identify the functional paths activated by i^c.

Register Access. In the register access stage, two sets of functional paths simply transfer the instruction operands from the register file. Activated functional paths are those used by exactly one of the instructions and can be readily identified from the two transition signatures $ts_1^{ra} = op_1^c \oplus op_1^p$ and $ts_2^{ra} = op_2^c \oplus op_2^p$, where $\oplus$ is the exclusive or operation.

Execute. In the execution stage of LEON3, all functional units share a single register for input operands. As a result, each functional unit performs its operation on the shared operands although the result of only one is registered. Therefore, the previous state of the active functional unit which, together with its current state, determines its sensitized paths, is the state induced by performing its operation on the operands of the previous instruction, even if it was a different type of instruction. To emulate this shared register scheme, we assume that the previous instruction i^p performs the same type of operation on its operands as the current instruction i^c. We will later show that this assumption can easily be implemented by inserting another instruction with source and destination registers of i^p and the opcode of i^c between the two instructions. Transition signatures are then defined based on the type of the instruction.

For logical instructions (`and`, `or`, etc.), each functional path is used if the corresponding bit in the result is 1 and the activated functional paths are those used by exactly one of the instructions, readily identified from the transition signature $ts^{exe} = r^c \oplus r^p$. Arithmetic instructions (`add`, `sub`, etc.) and memory access instructions (`ld` and `st`, which behave similar to `add`) have functional paths corresponding the carry chains of the addition they perform in the execute stage. Consider the multi-bit addition, $s = a + b$ in which $s_i = a_i \oplus b_i \oplus c_i$ where c_i denotes the input carry to ith bit. We can find carry bits by rewriting this as $c_i = a_i \oplus b_i \oplus s_i$. A carry chain from bit i to bit j ($i < j$) is used when $c_i = 0$, $c_{i+1} = ... = c_j = 1$, and $c_{j+1} = 0$. Therefore, we define two transition signatures $ts_1^{exe} = op_1^p \oplus op_2^p \oplus r^p$ and $ts_1^{exe} = op_1^c \oplus op_2^c \oplus r^c$. The activated functional paths are those used by exactly one of the additions.

Memory Access and Write-Back. The functional paths used in the memory access and write-back stages are determined by instruction results. Therefore, activated functional paths can be readily identified from the two transition signatures $ts^{mem} = ts^{wb} = r^c \oplus r^p$. Note that while the activation patterns are the same, functional paths of `ld` and `st` instructions in the memory stage are different from those of other instructions.

B. Source Code Instrumentation

In this section, we describe a source code instrumentation technique that extracts and stores the transition signatures and implements our timing speculation scheme. We explain the details using the example in Fig. 2 which shows how each basic block in the CFG is instrumented.

Incoming Edge Basic Block. A basic block is added along each incoming edge. Lines 1-2 call the function `bb_in` with the parameter `bb_id` that identifies the basic block. This function reads the error rate and is responsible for setting the frequency for the basic block. It also loads the control delays corresponding to the incoming edge into a pre-specified array to be used for estimating instruction delays. In lines 3-5, `r1`, `r2`, and `r3` represent any three regular registers not read or written in the basic block. These registers which are called *working registers* are copied into three Ancillary State Registers (ASRs). ASRs are a set of 16 registers provided by SPARC architecture for profiling and testing purposes.

Outgoing Edge Basic Block. A basic block is added along each outgoing edge. In lines 27-29, the working registers are restored to their original values. Lines 30-31 call the function `bb_out` with the basic block index which (i) identifies the activated functional paths using extracted transition signatures, (ii) uses them to estimate data delays, (iii) reads control delays and calculates instruction delays, and (iv) computes instruction error probabilities, the expected error rate, and speculation speedup.

Instrumented Basic Block. The original basic block is replaced by another basic block that identifies and stores the transition signatures of all its instructions. Lines 6-26 show the instructions that replace a `ld [%l1+%l2], %l3` instruction. We chose a load instruction to explain the instrumentation technique as it is the most complex type of instruction for transition signature extraction and shows all instrumentation code details . Lines 6-7 move operands of the previous instruction into `r1` and `r2`. Instrumentation codes of all instructions store the operands and the result of the the current instruction in `asr1`, `asr2`, and `asr3` (lines 16, 17, and 23). Lines 8-11 extract and store transition signatures for the register access stage (i.e. ts_1^{ra} and ts_2^{ra}). Line 12 emulates the shared operand register scheme by performing the operation of the current instruction in the execute stage on the operands of the previous instruction. Lines 13-15 then identify and store one of the execution stage transition signatures ts_1^{exe}. Operands of the current instruction are stored in `asr1`, `asr2` in lines 16-17 before the second transition signature of the execute stage, ts_2^{exe}, is extracted in lines 18-21. Line 22 is

Incoming Edge Basic Block

```
1.      set     %o0, bb_id
2.      call    bb_in; nop
3.      wr      %r1, %asr4
4.      wr      %r2, %asr5
5.      wr      %r3, %asr6
```

Instrumented Basic Block

```
...
6.      rd      %asr1, %r1
7.      rd      %asr2, %r2
8.      xor     %l1, %r1, %o0
9.      call    save_sig; nop
10.     xor     %l2, %r2, %o0
11.     call    save_sig; nop
12.     add     %r1, %r2, %r3
13.     xor     %r1, %r2, %o0
14.     xor     %r3, %o0, %o0
15.     call    save_sig; nop
16.     wr      %l1, %asr1
17.     wr      %l2, %asr2
18.     add     %l1, %l2, %r3
19.     xor     %r1, %r2, %o0
20.     xor     %r3, %o0, %o0
21.     call    save_sig; nop
22.     ld      %l1, %l2, %l3
23.     rd      %asr3, %r3
24.     xor     %l3, %r3, %o0
25.     call    save_sig; nop
26.     wr      %l3, %asr3
...
```

Outgoing Edge Basic Block

```
27.     rd      %asr4, %r1
28.     rd      %asr5, %r2
29.     rd      %asr6, %r3
30.     set     %o0, bb_id
31.     call    bb_out; nop
```

Fig. 2: Instrumentation code of basic block and load instruction as an example.

the original load instruction executed to ensure that behavior of the instrumented program does not change. Finally, the last pair of transition signatures, ts^{mem} and ts^{wb} are extracted and stored in lines 23-25 and the result of the instruction is stored in `asr3` in line 26. All arrays used for storing variables including frequencies, error rates, etc. are maintained as global variables as are functions `bb_in`, `save_sig`, and `bb_out` which are written in C and linked with the instrumented program. We implement the instrumentation technique in C++ and recompile the instrumented codes to produce executables. To perform the simulation, the instrumented program can be run on any machine that implements the ISA.

V. Experimental Results

In this section, we evaluate the timing speculation strategies proposed in this paper. Note that experiments for validating our error model were presented in Section III.

A. Experimental Setup

Synthesis and Static Timing Analysis. The design was synthesized on the 45nm TSMC technology targeting the typical-case corner $(TT, 0.9V, 25C)$. We set the frequency of our baseline system to 718MHz using SSTA at $(0.81V, 25C)$, guardbanding for a 10% voltage droop. For a fair comparison, we assume a fixed supply voltage of $0.81V$ for the timing speculative systems studied. This ensures that performance improvements accurately reflect the ability of the speculation strategies to track data variations.

Error Detection and Correction. We assume that error detection and correction circuits guarantee correct execution. We adopt a conservative error correction mechanism known as instruction replay at half-frequency. When a timing error is detected, the frequency is halved, the pipeline is flushed, and the errant instruction is reissued, resulting in a 24 cycle recovery penalty for our 6-stage pipeline.

Dynamic Frequency Tuning. Similar to a LEON3-based 45nm resilient Intel research processor [12], we consider a clock generator that uses phase-locked loop (PLL) based on the one in the 45nm Intel Core i7 microprocessor [13] which provides fine-grain frequency tuning in less than 2 cycles, which we consider as the penalty of each frequency change.

Power and Area Overheads. Implementing these adaptive clocking and error detection and correction schemes on a processor similar to LEON3 has been shown to incur a power and area overhead of less than 0.9% and 3.8%, respectively [12].

B. Speculation Strategies

Before we can evaluate the performance of our speculation scheme, we need to tune the design parameters described in Section II.B. We selected 12 applications from MiBench benchmark suite for our study. We used the small datasets of benchmark applications for tuning and the large datasets for performance evaluation. In all experiments, throughput values have been normalized to the throughput of the error-free guardbanded design.

1) Tuning N_B: This parameter which specifies the number of basic blocks for which timing speculation is performed determines the number of entries in the timing speculation table. For this evaluation, the other parameter is set to its default value, $N_S = 0$. Fig. 3 shows normalized throughput as N_B is increased by powers of two from 2 to 128. From the figure, it can be seen that maximum throughput is achieved for $N_B = 32$ or $N_B = 64$ depending on the application. A larger N_B increases throughput by improving the accuracy of frequency predictions. However, it also increases the number of frequency changes which incur a 2-cycle penalty each and limit throughput increase. Based on these results, we consider $N_B = 32$ or $N_B = 64$ for now.

2) Tuning N_S: This parameter specifies how many times a basic block skips frequency prediction and reuses the previous frequency before a new prediction is made. Since a larger N_S reduces the number of frequency changes, it could allow for a larger N_B as well. We then consider both $N_B = 32$

or $N_B = 64$ for this experiment. Fig. 4 shows normalized throughput as N_S is increased from 0 to 3. From the figure, it can be seen that the best value for N_S is highly dependent on the application. For example, the performance of `patricia` and `stringsearch` is significantly better for $N_S = 0$. This indicates a highly variable timing behavior which requires more frequent predictions to achieve high accuracy. In contrast, `bitcount`, `crc32`, and `dijkstra` exhibit highly predictable timing behaviors which allows for less predictions. It is interesting to note that when N_S is not 0, $N_B = 64$ performs better than $N_B = 32$ by limiting the number of frequency changes. Based on these experiments, we select the first two parameters, $N_B = 64$ and $N_S = 1$, for the next experiments.

3) Performance Evaluation: Using the tuned parameters, we evaluated the performance of our speculation scheme. Fig. 5 shows the results of our experiments for three timing speculative systems. Our speculation scheme is denoted by **Proposed**. The conventional approach, denoted by **Razor** in the figure, periodically records the global error rate and compares it to a threshold value. For a more conservative comparison, we chose the sampling frequency and error rate threshold values that maximized the performance on average. The **Oracle** strategy represents an ideal system that precisely predicts all instruction delays and instantly sets the frequency to the largest value that causes no timing errors for each basic block. We obtained the throughput of this system by simply summing the effective delays of all executed basic blocks. Effective delay of a basic block is the maximum of its instruction delays.

The figure shows that our proposed scheme consistently outperforms the conventional approach. On average, **Oracle** improves throughput of the guardbanded design by 143%. But the conventional approach achieves a 31.1% improvement, realizing less than 22% of the potential gains. While the strategies introduced in this paper achieves a throughput improvement of 50.9%, more than 64% of the potential gains remains untapped. This points to the significant opportunities for improving system performance with timing speculation.

VI. Conclusion

Using the efficient simulation framework described in this paper, we studied the opportunities provided by timing speculation for improving system performance and found that current methods realize only a fraction of the potential speedup. We proposed a timing speculation scheme that attains more performance gains by improving the quality of frequency predictions. Our timing speculation method limits the scope of speculation both in time and space. Spatially, we argued that frequency tuning should be directed by software and performed at the basic block level where instruction sequence is fixed and predictions are likely to be more accurate. Temporally, we showed that the most recent history of errors is a better predictor of timing behavior. Finally, we proposed to dynamically tune the frequency using an optimization algorithm instead of a controller. Together these strategies achieved a throughput improvement of 50.9%.

Acknowledgment

This material is based upon the work supported by the National Science Foundation's Variability Expedition in Computing under Award No. 1029783. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: circuit-level correction of timing errors for low-power operation," *Micro, IEEE*, vol. 24, no. 6, pp. 10–20, Nov 2004.

[2] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," *Comp. Arch. News*, vol. 38, no. 3, pp. 497–508, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1816026

[3] M. Choudhury and K. Mohanram, "Approximate logic circuits for low overhead, non-intrusive concurrent error detection," in *Design, Automation and Test in Europe, 2008. DATE '08*, March 2008, pp. 903–908.

[4] V. Kozhikkottu, S. Dey, and A. Raghunathan, "Recovery-based design for variation-tolerant socs," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 826–833. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228510

[5] P. Ndai, N. Rafique, M. Thottethodi, S. Ghosh, S. Bhunia, and K. Roy, "Trifecta: A nonspeculative scheme to exploit common, data-dependent subcritical paths," *Very Large Scale Integration (VLSI) Systems, IEEE Trans. on*, vol. 18, no. 1, pp. 53–65, Jan 2010.

[6] S. Roy and K. Chakraborty, "Predicting timing violations through instruction-level path sensitization analysis," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 1074–1081. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228555

[7] J. Xin and R. Joseph, "Identifying and predicting timing-critical instructions to boost timing speculation," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 128–139. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155636

[8] G. Hoang, R. B. Findler, and R. Joseph, "Exploring circuit timing-aware language and compilation," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 345–356. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950405

[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, Dec 2001, pp. 3–14.

[10] O. Assare and R. Gupta, "Timing analysis of erroneous systems," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES '14. New York, NY, USA: ACM, 2014, pp. 7:1–7:10. [Online]. Available: http://doi.acm.org/10.1145/2656075.2656101

[11] A. Gaisler, "Tsim erc32/leon simulator." [Online]. Available: http://www.gaisler.com/cms/

[12] K. Bowman, J. Tschanz, S. Lu, P. Aseron, M. Khellah, A. Raychowdhury, B. Geuskens, C. Tokunaga, C. Wilkerson, T. Karnik, and V. De, "A 45 nm resilient microprocessor core for dynamic variation tolerance," *Solid-State Circuits, IEEE Journal of*, vol. 46, no. 1, pp. 194–208, Jan 2011.

[13] N. Kurd, P. Mosalikanti, M. Neidengard, J. Douglas, and R. Kumar, "Next generation intel core micro-architecture (nehalem) clocking," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 4, pp. 1121–1129, April 2009.

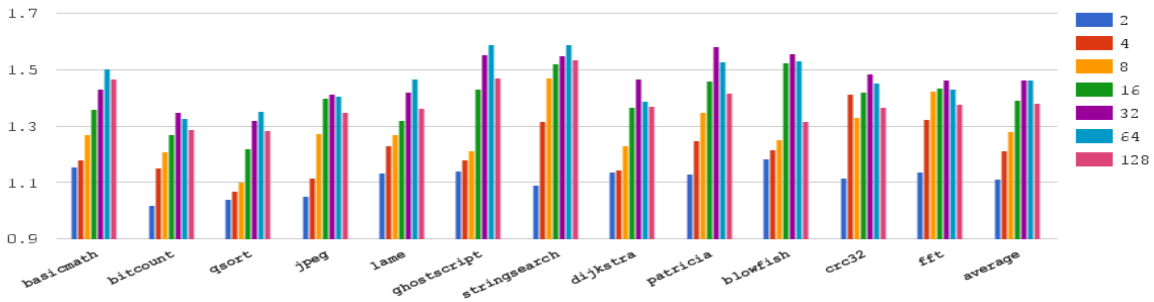
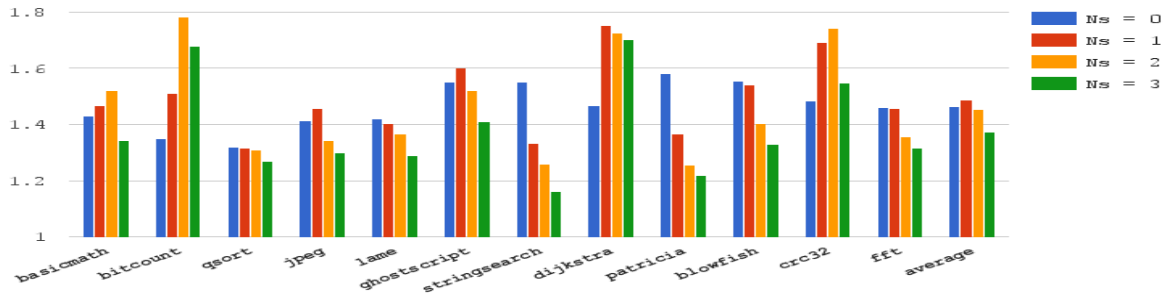
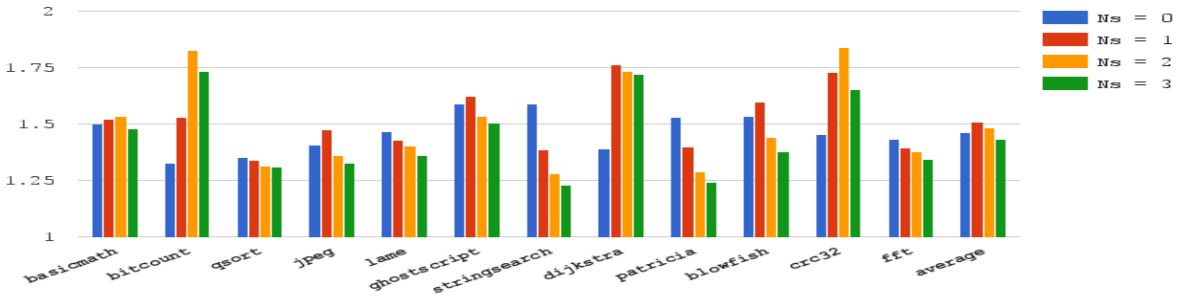


Fig. 3: Tuning N_B. Normalized throughput as N_B is increased from 2 to 128.



(a) $N_B = 32$



(b) $N_B = 64$

Fig. 4: Tuning N_S. Normalized throughput as N_S is increased from 0 to 3.

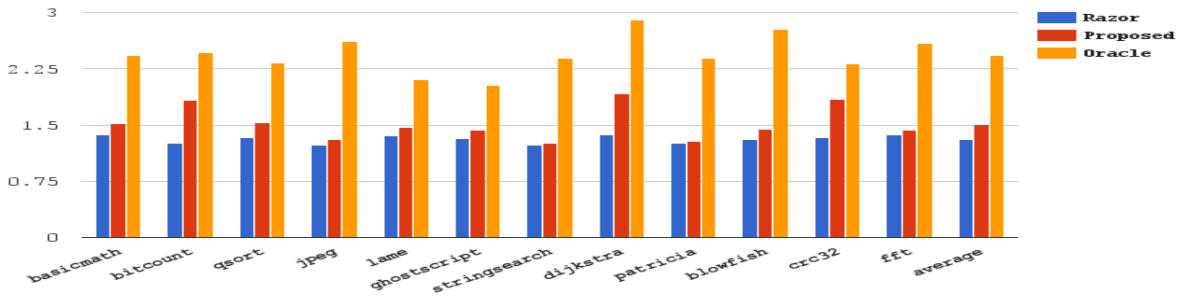


Fig. 5: Normalized throughput of our timing speculation scheme.