BandiTS: Dynamic Timing Speculation Using Multi-Armed Bandit Based Optimization

Jeff (Jun) Zhang Siddharth Garg
Department of Electrical and Computer Engineering, New York University, Brooklyn, New York, 11201
Email: {jeffjunzhang, sg175}@nyu.edu

***Abstract*—Timing speculation has recently been proposed as a method for increasing performance beyond that achievable by conventional worst-case design techniques. Starting with the observation of fast temporal variations in timing error probabilities, we propose a run-time technique to dynamically determine the optimal degree of timing speculation (i.e., how aggressively the processor is over-clocked) based on a novel formulation of the dynamic timing speculation problem as a multi-armed bandit problem. By conducting detailed post-synthesis timing simulations on a 5-stage MIPS processor running a variety of workloads, the proposed adaptive mechanism improves processor's performance significantly comparing with a competing approach (about 8.3% improvement); on the other hand, it shows only about 2.8% performance loss on average, compared with the oracle results.**

1. Introduction

Traditional integrated circuits (ICs) are designed to meet worst-case timing guarantees. Timing speculation (TS) is a new design approach that leverages the fact that the longest path of a circuit is rarely exercised at run-time. The idea is to clock a circuit at a frequency higher than its nominal frequency, and dynamically detect/correct any resulting timing errors. TS ensures faster execution in the common case (no timing errors), while guaranteeing correctness when timing errors do occur. Intel has recently demonstrated a timing-speculative microprocessor test chip, which enables more than 30% throughput gain over the conventional design [1].

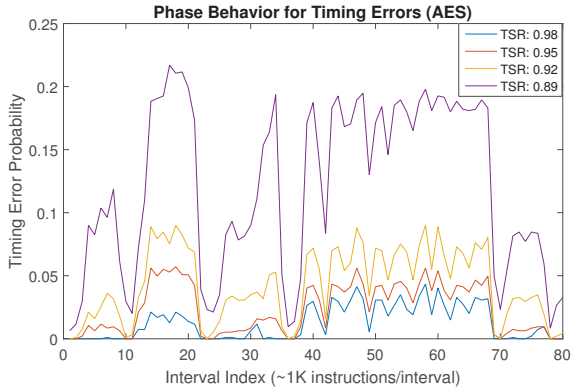

Figure 1: Phase behavior in timing errors across 80 intervals of AES benchmark.

Timing error probability increases monotonically with the amount of timing speculation. We refer to the ratio of TS frequency to the nominal frequency as the TS ratio. The more timing speculation (i.e., frequency increase) we perform, the higher error penalty we pay. Beyond a certain point, TS can hurt the performance because the overhead would be much greater than the performance gain. Thus, the key problem that motivates this paper is how to control the amount of timing speculation judiciously and dynamically, in order to achieve the best possible performance gain. Ideally, if we had oracle access to the timing error probability of a program in advance, we could figure out the optimal TS ratio easily. Unfortunately, to obtain this information online (i.e., at run-time) is not practical. Surprisingly, this problem has received little attention to date, despite the promise of the underlying TS architectures [2]–[4].

To determine the optimal TS ratio online, we need precise estimates of timing error probabilities at different TS ratios in the future. However, as shown in Fig. 1, the timing error probability can change drastically even within thousands of instructions, making it hard to predict. In order to adapt to the quick changes in timing error probability, the TS ratio needs to be adjusted frequently. However, this makes it harder to use traditional sampling based approach [5], [6] that are commonly used for dynamic control of voltage and frequency.

In this paper, we propose a new approach to the dynamic TS problem based on a reduction to the Multi-Armed Bandit (MAB) problem [7]. MAB is a classical problem in optimal control and reinforcement learning, where a "gambler" has to decide the best among multiple choices ("arms") that provide uncertain rewards. The solution to the problem attempts to resolve the trade-off between "exploration" (trying different arms) and "exploitation" (picking the best one seen so far). As we show, dynamic TS can be thought of as a MAB problem where each arm corresponds to a TS ratio, and stochastic rewards relate to (low) error probabilities.

By performing detailed post-synthesis timing simulations on a 5-stage MIPS processor, our results show that our proposed approach, which we refer to as BandiTS, outperforms the baseline approach by about 8.4% on average in terms of performance, and the regret of BandiTS on all benchmarks is only about 2.8% on average, comparing to the oracle results. Moreover, BandiTS algorithm is simple and practical to implement in hardware.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

2. Related Work

Many previous works on TS aim to increase single-thread performance through design and architectural optimizations [2]–[4]. Several low-overhead error recovery mechanisms are proposed, including instruction replay [8], counterflow pipelining [2], Bubble Razor [9], *etc*. Kruijf et al. [10] formulate a mathematical model for analyzing different TS techniques, and validate this model experimentally. Recently, several circuit level optimizations are proposed for TS [11], [12], which can further improve the circuit energy efficiency. In terms of dynamic TS, Xin et al. [13] propose a run-time mechanism to adapt the TSR by sampling based estimation of timing error probabilities. Recently, Yasin et al. [5] propose a data-dependent dynamic TS method for multi-core processors, which exploits spatial timing error heterogeneity from one core/thread to another. They too use a sampling based method to estimate error probabilities on the fly. In this paper we show that the sampling method works poorly, and instead propose a new algorithm based on multi-armed bandit optimization.

On a related note, dynamic TS can be thought of as an enhancement to the conventional dynamic voltage/frequency scaling (DVFS). In DVFS, the frequency of a core is set according to the worst case for any given voltage level. Our dynamic TS technique in this paper is orthogonal to DVFS and as such, dynamic TS can be performed synergistically with DVFS at any operating point for the additional gain.

3. Proposed Technique

3.1. Phase Behavior in Timing Errors

Sherwood et al. [14] were the first to explicitly note phase behaviour in program execution. In their paper, they investigated the program's phase behavior in terms of metrics such as cache misses, energy consumption, average instructions per cycle, etc. However, timing errors depend not only on the mix of instructions being executed but, more critically, on the actual *data* (i.e., operands) of each instruction. Therefore, one would expect to see even faster variations in timing errors than other architectural events (cache misses, for example) that do not depend on the actual data being read/written. This is indeed what we observe.

Fig. 1 shows error probabilities for four different TS ratios across 80 program intervals of AES. For any specific TS ratio, we can clearly see that the error probability changes with the intervals dramatically, even in the space of a few thousand instructions. This observation motivates the rest of our paper: can we intelligently react to and leverage those phase behaviors quickly while performing TS?

3.2. Adaption Approach

3.2.1. System Model

In this paper, we consider a processor equipped with the error detection and recovery mechanism as proposed in [2].

We denote the *nominal clock period* of the processor as t^{nom}, at which the processor is guaranteed to operate *without* any timing errors. For TS, the processor can operate at a clock period t^{clk} smaller than its nominal value.

Timing speculation ratio is defined as $r = \frac{t^{clk}}{t^{nom}}$, where $r \in R = \{r_1 = 1, r_2, ..., r_K\}$, is picked from K discrete levels in the descending order. *The timing error probability* is defined as a function of r, $p^{err} = err(r)$.

We can compute the *expected seconds per instruction (ESPI)* based on the model proposed by [10] for processors with fine-grained error recovery mechanisms like Razor, as follows:

$$ESPI = t^{nom} p^{err} C^{penalty} + t^{clk} CPI^{base}, \quad (1)$$

where CPI^{base} is the baseline clock cycles per instruction in the absence of errors, while $C^{penalty}$ is the recovery penalty. Here we assume, during the error recovery, the processor will run at its nominal clock period, which guarantees no timing errors are incurring during recovery. The *execution time* for running N instructions of a program interval is defined as:

$$t^{exec} = N * ESPI. \quad (2)$$

3.2.2. MAB Formulation

The multi-armed bandit (MAB) problem [7] has been studied extensively in reinforcement learning, modeling online decisions under uncertainty in a setting in which a gambler simultaneously attempts to acquire new knowledge and to optimize decisions based on the existing knowledge. The gambler is presented with multiple "levers" with stochastic rewards. The gambler iteratively plays one lever in each round and evaluates the associated reward. The goal is to maximize the total rewards over multiple pulls (in expectation). The more times a gambler pulls a lever, the more accurately he can estimate its average reward, but pays the cost of possibly pulling sub-optimal levers. This is referred to as the exploration-exploitation trade-off.

To reduce our problem to the multi-armed bandit, we treat every TS ratio as a potential "lever" that can be tried at run-time. For every pull, we get an estimate of the p^{err} for that TS ratio with corresponding expected cost t^{exec}. More formally, our version of the stochastic K-armed bandit can be seen as a set of K levers with error probability $P = \{err(r_1), err(r_2), ..., err(r_K)\}$, being associated to the expected cost $T = \{t_1^{exec}, t_2^{exec}, ..., t_K^{exec}\}$.

At each pull i (i.e., every N instructions of a program interval), the player selects an arm $j(i) \in K$, and receives a cost $t_{j(i)}^{exec} \in T$. The player's goal is to incur as little cost as possible over total M pulls, that is:

$$min_{j(i)} \sum_{i \in [1,M]} t_{j(i)}^{exec}. \quad (3)$$

To measure the performance of the results, the cumulative regret ρ after M pulls is defined as the difference between the sum of the incurred cost and, the cost associated to an optimal strategy, i.e.,

$$\rho = \sum_{i \in [1,M]} (t_{j(i)}^{exec} - t(i)^*), \quad (4)$$

where $t(i)^*$ is the ideal minimal cost for each pull i, assuming access to an oracle.

3.2.3. BandiTS

In this work, we adopt one of the simplest and most widely used bandit approaches, the ϵ-greedy strategy. Vermorel et al. and Kuleshov et al. [15], [16] have both shown that simple heuristics such as ϵ-greedy obtain empirically lower regret compared to more sophisticated policies.

Algorithm 1: BandiTS optimization procedure

```
Algorithm BandiTS(K,M,ε)
Initialize the estimated P̂, T̂ by pulling each arm once;
cost ← Σ_{i∈[1,K]} t_i^exec;
for i ∈ [1, M − K] do
    Flip a coin with ε probability to be head,
    and 1 − ε probability to be tail;
    if head up then
        Pull a random arm j;
    else
        Pull the arm j ← argmin_j t̂_j^exec;
    end
    cost ← cost + t_j^exec;
    Update the estimated êrr(r_j) and t̂_j^exec for arm j;
end
return cost;
```

Our proposed BandiTS procedure is shown in Algorithm 1. It iteratively makes the decision based on estimators $\hat{P}$ and $\hat{T}$ that are computed from prior pulls to predict the expected future cost. BandiTS chooses a random arm with probability ϵ (exploration), and otherwise chooses the arm with the lowest estimated mean cost (exploitation). The parameter ϵ is a tunable knob for the trade-off between exploration and exploitation.

In this policy, $n_j(i)$ stores the number of times that the arm j has been selected up to the time step i. $\widehat{err}(r_j)$ of the selected arm j is updated for step i as follows:

$$\widehat{err}(r_j) = \frac{\widehat{err}(r_j) \times n_j(i-1) + err_i(r_j)}{n_j(i-1)+1}, \qquad (5)$$

where $err_i(r_j)$ is the actual error probability by choosing TS ratio r_j for N instructions of program interval i. That is, after we run these instructions in step i, the error probability $err_i(r_j)$ is known to us. $\widehat{t}_j^{exec}$ is then updated according to Equation (1), (2) and (5).

4. Evaluation

4.1. Experiment Setup

We evaluate BandiTS on TigerMIPS, a 32-bit 5-stage processor that supports the MIPS instruction set [17], and take CHStone as the set of benchmark applications [18].

Using Quartus II, we first synthesize the TigerMIPS RTL to obtain the post-placement gate-level netlist and the corresponding standard delay file (SDF) for each stage of the pipeline. Then, we only back-annotate the execution stage of the MIPS processor with its SDF file, and conduct a detailed post-synthesis timing simulation to get each instruction's execute stage delay. The delay traces are used as input to an implementation of our BandiTS approach in C. At each clock cycle, we feed the simulator with one instruction. To model the timing errors, we add a penalty of five cycles [5].

4.2. Simulation Results

We compare our proposed BandiTS with several baselines:

- **Nominal**: the processor executes at its nominal clock period without any timing speculation applied.
- **Sampling**: each program interval is divided into "sampling" and "running" phase. During the sampling stage, the processor runs in turn at each TS ratio to estimate the corresponding error rate, and chooses the best for the running stage [5], [13]. The sampling phase is set to be 10% of every interval.
- **BanditOracle**: assuming access to an oracle, the processor is executed at the optimal TS ratio in each control interval. BanditOracle provides us a lower bound to compare BandiTS with, although this bound is unachievable in practice.

Before discussing our results, we investigate the role of two important parameters, ϵ and interval size on the results.

Role of ϵ: To examine the role of ϵ in the BandiTS, we simulate three ϵ values, namely 0.1, 0.05, 0.01, on each benchmark. For all benchmarks, we find that with $\epsilon = 0.01$, the BandiTS achieves the best performance. Due to the space limit, we only show the results with $\epsilon = 0.01$ in Figure 2. The results for BandiTS are averaged over 1000 runs.

Role of the interval size: Figure 2 shows the execution time of our BandiTS on each benchmark for different control interval sizes. The results are normalized to the nominal baseline. Note that the performance of BandiTS improves as the interval size becomes smaller since it is able to better adapt to the phase behaviour.

We observe that, (1) Both Sampling and BandiTS obtain performance improvement (at least 25%) over the nominal baseline at their best settings in Figure 2. (2) BandiTS beats Sampling method in all benchmarks in Figure 2. Also, the results of BandiTS are more stable than the Sampling method. (3) In Figure 3, we select the best control interval size for BandiTS and for the Sampling method separately. On average, BandiTS outperforms the Sampling approach by about 8.4% in terms of performance. (4) On the other hand, BandiTS provides near-optimal results comparing to the BanditOracle. The sub-optimality of BandiTS on all benchmarks is only about 2.8% on average.

5. Conclusion

In this paper, we have designed a new dynamic TS strategy, BandiTS, to adapt to temporal variations in timing error probability during a program's execution. Our analysis on a

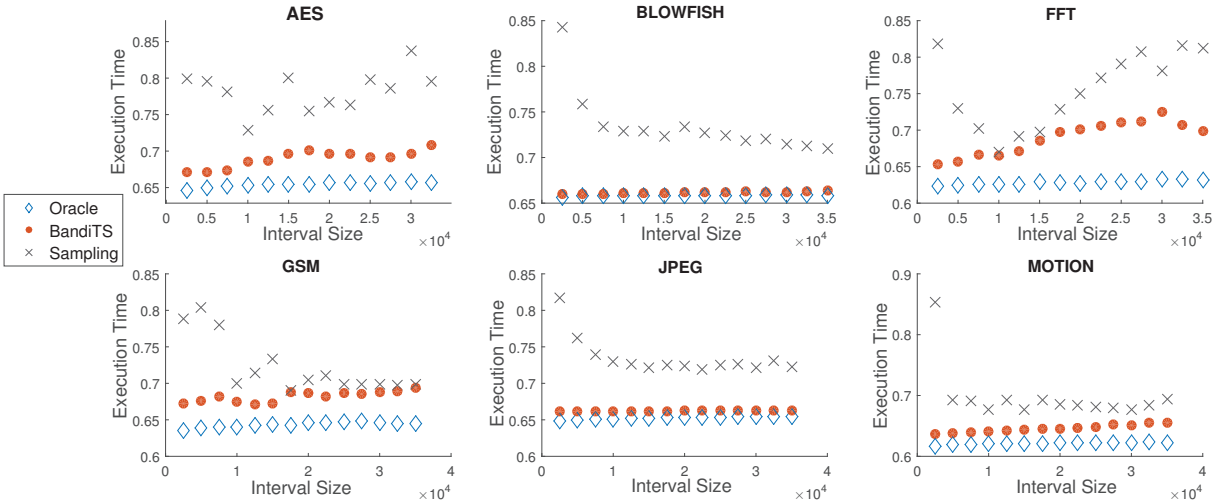

Figure 2: Execution time for CHStone benchmarks (ϵ is 0.01 for BandiTS), normalized to the Nominal baseline.

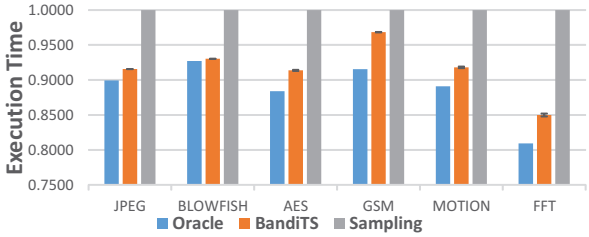

Figure 3: Best BandiTS (ϵ is: 0.01, control size: 500 ins., average on 1000 experiments) versus best Sampling (sampling size: 2000 ins.). Results are normalized to the Sampling.

range of benchmarks shows that the proposed BandiTS strategy outperforms state-of-the-art sampling based approaches in terms of program execution time.

References

[1] K. A. Bowman, J. W. Tschanz, N. S. Kim, J. C. Lee, C. B. Wilkerson, S. L. L. Lu, T. Karnik, and V. K. De, "Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 1, pp. 49–63, Jan 2009.

[2] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," in *Proceedings of 36th IEEE/ACM International Symposium on Microarchitecture*, Dec 2003, pp. 7–18.

[3] B. Greskamp and J. Torrellas, "Paceline: Improving single-thread performance in nanoscale cmps through core overclocking," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, 2007, pp. 213–224.

[4] B. Greskamp, L. Wan, U. R. Karpuzcu, J. J. Cook, J. Torrellas, D. Chen, and C. Zilles, "Blueshift: Designing processors for timing speculation from the ground up." in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, Feb 2009, pp. 213–224.

[5] A. Yasin, J. Zhang, H. Chen, S. Garg, S. Roy, and K. Chakraborty, "Synergistic timing speculation for multi-threaded programs," in *Proceedings of the 53nd DAC*. ACM, 2016, p. 183.

[6] W. R. Dieter, S. Datta, and W. K. Kai, "Power reduction by varying sampling rate," in *Proceedings of the international symposium on Low power electronics and design*. ACM, 2005, pp. 227–232.

[7] P. Whittle, "Multi-armed bandits and the gittins index," *Journal of the Royal Statistical Society.*, pp. 143–149, 1980.

[8] S. Das, C. Tokunaga, S. Pant, W.-H. Ma, S. Kalaiselvan, K. Lai, D. M. Bull, and D. T. Blaauw, "Razorii: In situ error detection and correction for pvt and ser tolerance," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 1, pp. 32–48, 2009.

[9] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. Harris, D. Blaauw, and D. Sylvester, "Bubble razor: An architecture-independent approach to timing-error detection and correction," in *International Solid-State Circuits Conference*. IEEE, 2012, pp. 488–490.

[10] M. De Kruijf, S. Nomura, and K. Sankaralingam, "A unified model for timing speculation: Evaluating the impact of technology scaling, cmos design style, and fault recovery mechanism," in *International Conference on Dependable Systems and Networks*. IEEE, 2010, pp. 487–496.

[11] Y. Liu, F. Yuan, and Q. Xu, "Re-synthesis for cost-efficient circuit-level timing speculation," in *Proceedings of the 48th DAC*. ACM, 2011, pp. 158–163.

[12] A. Kapare, H. Cherupalli, and J. Sartori, "Automated error prediction for approximate sequential circuits," in *Proceedings of the ICCAD*. ACM, 2016.

[13] 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*, 2011, pp. 128–139.

[14] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *Micro*, vol. 23, no. 6, pp. 84–93, 2003.

[15] J.-Y. Audibert, R. Munos, and C. Szepesvári, "Exploration–exploitation tradeoff using variance estimates in multi-armed bandits," *Theoretical Computer Science*, vol. 410, no. 19, pp. 1876–1902, 2009.

[16] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *arXiv preprint arXiv:1402.6028*, 2014.

[17] S. Moore and G. Chadwick. The tiger "mips" processor. [Online]. Available: http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html

[18] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems*, 2008, pp. 1192–1195.