Synergistic Timing Speculation for Multi-threaded Programs

Atif Yasin¹, Jeff (Jun) Zhang², Hu Chen¹, Siddharth Garg², Sanghamitra Roy¹ and Koushik Chakraborty¹

¹BRIDGE Lab, Electrical and Computer Engineering, Utah State University

²Department of Electrical and Computer Engineering, New York University

¹atifyasin2@gmail.com, hu.chen@aggiemail.usu.edu, {sanghamitra.roy, koushik.chakraborty}@usu.edu

²{jeffjunzhang, sg175}@nyu.edu

ABSTRACT

In this paper, we address the problem of timing speculation for multi-threaded workloads executing on a multi-core processor. Our approach is based on a new observation — heterogeneity in path sensitization delays across different threads in multi-threaded programs. Leveraging this heterogeneity, we propose Synergistic Timing Speculation (SynTS) to jointly optimize the energy and execution time of multi-threaded applications. In particular, SynTS uses a sampling based online error probability estimation technique, coupled with a polynomial time algorithm, to optimally determine the voltage, frequency and the amount of timing speculation for each thread. Our experimental evaluations, based on detailed cross-layer simulations, demonstrate that SynTS reduces energy delay product by up to 21%, compared to existing timing speculation schemes.

1 Introduction

Timing speculation is a promising approach to increase the processor performance and energy efficiency [9]. Under timing speculation, an integrated circuit is allowed to operate at a speed faster than its slowest path—the critical path. It is based on the empirical observation that these critical path delays are rarely manifested during program execution. Consequently, as long as the processor is equipped with an error detection and recovery mechanism, its performance can be increased and/or energy consumption reduced beyond that achievable by conventional operation.

While many past works have dealt with timing speculation within a single core, in this work, we uncover a new direction by exploring *timing speculation for a multi-core processor executing a parallel, multi-threaded application*. Through a rigorous circuit-architectural analysis, we observe that during the execution of a multi-threaded program, there is a significant variation in circuit delay characteristics across different threads. Consequently, under timing speculation (e.g., a higher operating frequency), some threads exhibit higher timing error¹ probabilities—referred as *timing speculation critical threads*. For the same amount of timing speculation (i.e., frequency increase), a timing speculation critical thread has a longer execution time than other threads, even

¹Timing error is seen when the incurred circuit delay exceeds the clock period.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

DAC '16, June 05–09, 2016, Austin, TX, USA

Copyright 2016 ACM. ISBN 978-1-4503-4236-0/16/06 ...\$15.00

DOI: <http://dx.doi.org/10.1145/2897937.2898102>.

Figure 1: Timing error probability with a normalized clock period for one barrier interval in the Radix benchmark.

if the thread execution times are balanced during nominal operation.

1.1 Motivational Example

Figure 1 shows an example of a timing speculation critical thread in the Radix benchmark from the SPLASH-2 benchmark suite (see Section 4 for our detailed methodology). In this example, Thread 0 consistently has the highest error probability with decreasing clock period, about 4× greater than the thread with the lowest error probability.

To exploit this intriguing thread-level heterogeneity in timing error characteristics, we propose to perform *synergistic timing speculation* for multi-threaded workloads by incorporating timing speculation criticality. We call our approach SynTS. Figure 2 explains, the opportunities of SynTS to substantially improve the system energy efficiency by exploiting heterogeneity in thread error probabilities.

We start with a scenario in which all four threads run at the nominal voltage and frequency without timing speculation. We will assume that the threads are racing to a barrier, and reach the barrier at the same time at the nominal voltage and frequency. This means that the threads are perfectly balanced—with perfect work distribution and perfect cache access latencies. In Step 1, we exploit timing speculation to reduce thread execution time by increasing the frequency (reducing clock period) for all threads at the expense of a small error probability. As shown in Figure 2, reducing the clock period by 24%, reduces execution time of Thread 0 by 7%, observed empirically. The execution times of threads 1, 2 and 3 are reduced even further, since they have lower error probabilities. Any further increase in clock frequency hurts performance since it causes an increase in timing errors and its associated recovery overheads in Thread 0, nullifying the positive affects of higher frequency.

At the end of Step 1, we note that although the threads all reach the barrier sooner than in the nominal case, Thread

Figure 2: Overview of the SynTS approach. The data here are generated based on the error probability curve in Figure 1. More details about the algorithm and experimental parameters are given in Section 3 and Section 4, respectively.

0 is now critical, i.e., it reaches the barrier last, while the other three threads have some slack.

In Step 2, we leverage the slack that has been created to reduce the voltage and potentially frequency of Threads 1, 2 and 3, thus reducing their energy consumption without hurting the application’s execution time. In this case, the voltage of all three threads is reduced to 0.9V. Overall, for this motivational example, *both the execution time and the energy consumption of the barrier interval reduce by 7%, each*. This example demonstrates the *dual* benefits in execution time and energy consumption—the significant potential of *synergistic* timing speculation that is not achievable by naively adapting existing single-core timing speculation approaches to the multi-core setting.

In summary, this example motivates the key research question that we answer in this paper: *how can we synergistically determine the optimal voltage, frequency, and as a result the error probability, for each thread/core, so as to jointly optimize for execution time and energy consumption?*

1.2 Paper Contributions

To the best of our knowledge, ours is the first work to deal with the problem of synergistic timing speculation for multi-threaded applications. To this end, we make the following novel contributions:

- Using an elaborate cross-layer methodology, we provide the first empirical evidence of heterogeneity in error probability of threads under timing speculation for SPLASH-2 benchmark applications. We propose SynTS, an approach that is aware of this heterogeneity and synergistically performs timing speculation for all threads to optimize the performance and energy.
- We propose a mathematical formulation for SynTS as a discrete optimization problem, and propose a polynomial-time algorithm that optimally solves this problem. Subsequently, we propose a practical online implementation of SynTS based on error probability sampling and our polynomial-time algorithm.
- Using rigorous circuit-architectural simulations, we demonstrate a significant improvement in energy-efficiency—up to 28% reduction in the energy delay product (EDP), compared to existing timing speculation schemes.

2 Related Work

Previous works most relevant to our work fall in two broad categories: (a) timing speculation architectures for single-core processors; and (b) conventional voltage/frequency scaling techniques for multi-threaded applications. The works in the first category aim at reducing the clock cycle time

to boost performance, and/or reduce the supply voltage to save energy. Many of these techniques are recovery-based, where rare timing-errors are detected and recovered using micro-architectural techniques [9, 5, 1]. Some recent works propose proactive techniques, where they anticipate an upcoming timing error before the clock edge, using various sensors embedded in the pipe stages [8, 13, 11]. Several other works advocate dynamic clock skewing [16, 18], in combination with timing speculation, for energy efficiency. All of these papers are focused on isolated processor components and/or a single-core pipeline, and do not address how to synergistically overclock or voltage scale multi-core processors.

The second category of works explores the use of conventional voltage/frequency scaling (i.e., without any timing speculation), to optimize the energy and execution time of multi-threaded applications. In these works, the criticality of threads is assessed from their individual execution latency variance due to specific architectural events (e.g., cache misses) or the balance of work among threads [12, 2, 15, 4]. None of these works address timing speculation for multi-threaded applications or exploit timing speculation criticality, which our work does for the first time.

3 SynTS Design

We describe our design of SynTS in three steps. First, we discuss our mathematical model for timing speculation on multi-core processors (Section 3.1). Then, we describe an ideal offline implementation of SynTS (Section 3.2), and finally present a practical online version (Section 3.3).

3.1 System Model

In this paper, we consider a multi-core processor consisting of M homogeneous cores, and a multi-threaded application executing on the processor with one thread per core (that is, the number of threads is also M). The cores are equipped to handle timing speculation, that is, they can both detect and recover from errors using schemes proposed in literature [9].

Each core can dynamically tune its voltage and clock frequency (or equivalently, clock period); a capability that is available in several commercial multi-core processors. The voltage of core i ($i \in [1, M]$) is $V_i \in \mathcal{V}$, picked from Q discrete voltage levels, i.e., $\mathcal{V} = \{V_1, V_2, \dots, V_Q\}$. For every voltage level $V \in \mathcal{V}$, there is a *nominal clock period* $t^{nom}(V)$ at which the core is guaranteed to operate *without* any errors.

To enable timing speculation, core i can operate at a clock period smaller than the nominal period. That is, the clock period of core i , t_i^{clk} , is a ratio r_i ($r_i \in [0, 1]$) of its nominal period; $t_i^{clk} = r_i t^{nom}(V_i)$.

We refer to r_i as the *timing speculation ratio* (TSR), and assume that it is picked from one of S discrete levels including 1. That is, $r_i \in \mathcal{R}$, where $\mathcal{R} = \{\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_S = 1\}$. Note that the TSR implicitly corresponds to the discrete clock frequency at which cores can operate.

For a given r_i , the error probability is given as $p_i^{err} = err_i(r_i)$. err_i is a decreasing function of r_i ; longer clock periods imply lower error probability. An example of the error probability function are shown in Figure 1. As observed before, the error probability function can *vary from one thread to another*, i.e., it is thread specific.

With these preliminaries, we can now model system performance and energy consumption. Our performance model is based on the one proposed by [10] for processors with fine-grained error recovery mechanisms like Razor. In particular, the seconds per instruction (SPI)² of thread i can be written as:

$$SPI_i = t_i^{clk} \left(p_i^{err} C^{penalty} + CPI_i^{base} \right) \quad (1)$$

where $C^{penalty}$ is the error recovery penalty (5 cycles for the Razor processor [10]) and CPI_i^{base} is the baseline clock per instruction of thread i in the absence of errors.

Our focus in this work is on parallel applications that use barrier synchronization. Without loss of generality, we will model application execution time for a single barrier interval; the total execution time can be obtained by summing over all barriers interval. In a given barrier interval, thread i executes N_i instructions. The execution time of the barrier phase, t_{exec} , is determined by the *last* thread to reach the barrier [2]:

$$t_{exec} = \max_{i \in [1, M]} N_i t_i^{clk} \left(p_i^{err} C^{penalty} + CPI_i^{base} \right) \quad (2)$$

Finally, the energy consumption of a thread, en_i , can be written as:

$$en_i = \alpha V_i^2 N_i \left(p_i^{err} C^{penalty} + CPI_i^{base} \right) \quad (3)$$

where α is the average switching capacitance of a core. The energy equation multiplies the energy consumed per clock cycle with the number of clock cycles, including the extra cycles introduced due to timing speculation. We note that although the model does not currently account for leakage power, it can be easily extended to do so. Furthermore, although our focus in this paper is exploring the *energy* versus execution time trade-offs, the proposed approach can be generalized to address power consumption as well.

3.2 Offline Optimization

We now describe our proposed synergistic timing speculation methodology. We begin by optimistically assuming that the workload characteristics, that is, each thread's error probability function, $err(r_i)$, is known in *advance*. In Section 3.3, we will relax this assumption and describe an *online* policy that estimates the error probability function for each thread on the fly.

Our goal is to determine the optimal voltage and clock period of cores in each barrier interval, so as to minimize a weighted sum of the total energy consumption and the barrier execution time. This can be formulated as the following optimization problem, **SynTS-OPT**,

$$\min_{V_i, r_i} \sum_{i \in [1, M]} en_i + \theta t_{exec} \quad (4)$$

²Inverse of instruction per second (IPS).

Algorithm 1: SynTS-Poly optimization procedure

```

1 Algorithm
   SynTS-Poly( $\mathcal{V}, \mathcal{R}, t^{nom}, err, C^{penalty}, CPI^{base}$ )
2   for  $i \in [1, M]$  do
3     for  $j \in [1, Q]$  do
4       for  $k \in [1, S]$  do
5          $p_i^{err} \leftarrow err_i(\mathcal{R}_k)$ ;
6          $t_i^{clk} \leftarrow t^{nom}(\mathcal{V}_j) \mathcal{R}_k$ ;
7          $t_{exec} \leftarrow N_i t_i^{clk} (p_i^{err} C^{penalty} + CPI_i^{base})$ ;
8          $en \leftarrow \alpha \mathcal{V}_j^2 N_i (p_i^{err} C^{penalty} + CPI_i^{base})$ ;
9         for  $l \in [1, M] \wedge l \neq i$  do
10           $en \leftarrow en + \minEnergy(l, t_{exec})$ ;
11        end
12         $cost_{ijk} \leftarrow en + \theta t_{exec}$ ;
13      end
14    end
15  end
16  return  $\arg \min_{ijk} cost_{ijk}$ ;
17 Procedure  $\minEnergy(i, t_{exec})$ 
18   for  $j \in [1, Q]$  do
19     for  $k \in [1, S]$  do
20        $en_{jk} \leftarrow \infty$ ;
21        $p_i^{err} \leftarrow err_i(\mathcal{R}_k)$ ;
22       if
          $N_i t^{nom}(\mathcal{V}_j) \mathcal{R}_k (p_i^{err} C^{penalty} + CPI_i^{base}) \leq$ 
          $t_{exec}$  then
23          $en_{jk} \leftarrow \alpha \mathcal{V}_j^2 N_i (p_i^{err} C^{penalty} + CPI_i^{base})$ ;
24       end
25     end
26   end
27   return  $\min_{jk} en_{jk}$ ;

```

such that $V_i \in \mathcal{V}$ and $r_i \in \mathcal{R}$ for all $i \in [1, M]$. θ is a designer-specified weighting factor that determines the importance of execution time vis-a-vis energy.

MILP Formulation SynTS-OPT is a discrete, non-linear optimization problem. We first reduce it to a mixed integer linear programming (MILP) problem, which we refer to as **SynTS-MILP**. To do so, we introduce binary variables x_{ijk} that are set to 1 if thread i runs at voltage level $\mathcal{V}_j$ and TSR $\mathcal{R}_k$, and 0 otherwise. The objective function can now be written as:

$$\min_{x, en, p^{err}, t^{clk}, t_{exec}} \sum_{i \in [1, M], j \in [1, Q], k \in [1, S]} x_{ijk} en_{ijk} + \theta t_{exec}, \quad (5)$$

subject to:

$$t_{exec} \geq N_i t_i^{clk} \left(p_i^{err} C^{penalty} + CPI_i^{base} \right) \quad \forall i \in [1, M], \quad (6)$$

$$t_i^{clk} = \sum_{j \in [1, Q], k \in [1, S]} x_{ijk} \mathcal{R}_k t^{nom}(\mathcal{V}_j) \quad \forall i \in [1, M], \quad (7)$$

$$p_i^{err} = \sum_{j \in [1, Q], k \in [1, S]} x_{ijk} err_i(\mathcal{R}_k) \quad \forall i \in [1, M], \quad (8)$$

$$en_{ijk} = N_i \left(p_i^{err} C^{penalty} + CPI_i^{base} \right) \alpha \mathcal{V}_j^2 x_{ijk} \quad \forall i \in [1, M], \quad (9)$$

and

$$\sum_{j \in [1, Q], k \in [1, S]} x_{ijk} = 1 \quad \forall i \in [1, M]. \quad (10)$$

Equation 6 constrains execution time of the barrier to be larger than that of each thread, while Equations 7, 8 and 9 compute the clock period, error probability and energy, respectively, in terms of the x_{ijk} variables. Equation 10 ensures that each thread gets assigned to only one voltage and frequency level. **SynTS-MILP** can be input to a standard MILP solver to obtain *optimal* voltage and frequency levels for each thread.

Polynomial-time Algorithm Solving a MILP problem is, however, not practical in an *online* setting since the run-time of MILP solvers scales poorly with the problem size. Fortunately, the specific form of **SynTS-MILP** lends itself to a polynomial-time solution shown in Algorithm 1 – a key contribution of our paper.

The intuition behind our algorithm, **SynTS-Poly**, is as follows. We iteratively demarcate each thread as the *critical* thread, i.e., the thread that has the longest execution time ($\mathcal{O}(M)$ iterations). For a critical thread, we try all combinations of voltage and timing speculation ratios; for each such combination, we obtain the thread’s execution time ($\mathcal{O}(QS)$ iterations). Then, for every other thread, we search for the *lowest energy* configuration that allows it to finish before or with the critical thread ($\mathcal{O}(MQS)$ iterations). These steps yield a pair of energy and execution time values for each thread, voltage and clock period combination (a total of MQS pairs). Of these, we return the configuration with the lowest weighted cost. The run-time of **SynTS-Poly** is *quadratic* in the number of threads/cores, voltage levels and timing speculation ratios.

LEMMA 3.1. *Algorithm 1 is guaranteed to return an optimal solution for SynTS-OPT.*

PROOF. Detailed proof is omitted for brevity, but we note that it depends on the fact that the non-critical threads only impact the energy component of the cost function. $\square$

3.3 Online Optimization

In practice, the error probability versus the clock period data is *not* available in advance, and must be estimated on-the-fly. We propose an online sampling based approach to address these practical constraints.

At the beginning of each barrier interval, each thread spends the first N_{samp} instructions in a *sampling phase*. During the sampling phase, all threads run at a fixed voltage $V_{samp} \in \mathcal{V}$, but at different clock periods. In particular, each thread spends $\frac{N_{samp}}{S}$ instructions at each of the S available frequency levels (or equivalently, timing speculation ratios), for the voltage level V_{samp} .

At the end of the sampling phase, we obtain an estimate of the error probability function, err_i , for each thread i . We refer to the estimated error probability function as $e\tilde{r}_i$. Note that although the error probability estimate is obtained for a single voltage level (V_{samp}), the error at any other voltage $V \in \mathcal{V}$ is estimated as $e\tilde{r}_i(\frac{t_{clk}^{clk}}{t_{nom}(V)})$.

Finally, the estimated error probability functions are provided as the input to **SynTS-Poly** algorithm (Algorithm 1), which returns optimized voltage and timing speculation ratios (i.e., clock periods) for each thread. The threads are then run at these optimized levels for the remaining barrier interval.

The number of instructions in the sampling phase, N_{samp} and the voltage at which cores execute in the sampling phase, V_{samp} , are both knobs in our online approach. Increasing N_{samp} provides more precise error estimates, but results in greater energy and execution time overheads during sampling. Increasing V_{samp} increases the energy but reduces

Table 1: Voltage versus Nominal clock period.

Vdd (V)	1.0	0.92	0.86	0.8	0.72	0.68	0.65
$t^{nom} (\times)$	1.0	1.13	1.27	1.39	1.63	2.21	2.63

the execution time overhead of sampling. Section 5 discusses how we set values for these parameters.

4 Evaluation Methodology

In this section, we describe our cross-layer methodology that we use to evaluate the proposed synergistic timing speculation techniques. Figure 3 gives an overview of our methodology. We now describe our methodology in detail.

Figure 3: Cross-layer methodology to profile delay characteristics.

Architectural Simulator We use the cycle-accurate Gem5 simulator [3] to model a 4-core Alpha processor. We use several Splash2-benchmarks representing a range of real world applications [17], on the simulator and extract cycle-by-cycle input vectors for each stage in the processor. These input vectors are used to drive the netlist in the circuit-level timing analysis, to estimate the actual propagation delay of each instruction during program execution. We run each benchmark for 4 barrier intervals, or to its completion, whichever comes first.

Timing Error Modeling Using Synopsys Design Compiler, we synthesize the Illinois Verilog Model [14] of the Alpha processor to obtain a gate-level netlist for each pipe stage. Next, by feeding cycle-by-cycle input vectors for each stage to its structural RTL netlist, we record the sensitized paths for each instruction. In this work, we focused on the execute pipe stage, specifically the SimpleALU. The propagation delays of gates on the sensitized paths are obtained from HSPICE simulation using the Predictive Technology Model (PTM) for the 22nm node [19]. Finally, to model the impact of voltage scaling on the propagation delay, we use HSPICE simulations of 22 nm ring oscillators and record the clock period versus voltage, as shown in Table 1.

Based on this cross-layer methodology, we obtain traces of propagation delays for each instruction, for different supply voltages. Now, for any given speculative clock period, we are able to identify which instructions induce timing failures. We replay the traces using the oneIPC model [7], and add a penalty of five clock cycles for error inducing instructions [10].

Benchmarks We use the above methodology to characterize the error probability functions of 5 SPLASH-2 benchmarks—FMM, Radix, LU, FFT and Ocean. Of these, FFT and Ocean have homogeneous error probabilities for all threads, for which conventional timing speculation and our approach would work just as well. In fact, the FFT error probabilities are high and do not permit *any* timing speculation. Hence, we report our results on FMM, Radix and LU.

Figure 4: Energy versus execution of FMM for offline versions of the SynTS, Per-core TS and No-TS approaches, normalized to the Nominal baseline. Each point corresponds to a different value of weight θ from Equation 4.

Figure 5: Same data as in Figure 4 for the Radix benchmark.

5 Experimental Results

We compare our proposed SynTS approach to several comparative schemes, as described below:

- **Nominal V/F (Nominal)**: each core runs at its nominal voltage and corresponding clock period, i.e., without any V/F scaling and without any timing speculation.
- **Optimal V/F without timing speculation (No-TS)**: each core runs at different voltage levels so as to minimize the weighted cost function of energy and execution time in Equation 4, but *without* any timing speculation. The No-TS baseline reflects existing approaches that attempt to balance workload variations between threads using DVFS as proposed by [12].
- **Per-core timing speculation (Per-core TS)**: each core leverages timing speculation, to *independently* minimize its own energy and execution time cost, as measured using Equation 4. Per-core TS serves as a best possible bound for single-core timing speculation techniques like Razor [9], since it has offline access to the error probability functions for each core.

We begin by presenting results for the offline version of SynTS, and then present results for a practical online implementation of SynTS.

5.1 Offline Optimization Results

We begin by comparing the above schemes above to SynTS in the offline setting; i.e., assuming that the error probability functions in each barrier interval are known in advance. Although such an offline approach cannot be implemented in practice, it allows us to quantify the best results that can be obtained from SynTS.

Figure 6: Actual and online estimated error probability versus timing speculation ratio for one barrier interval in the Radix and FMM benchmark.

To compare the energy vs. execution time trade-offs that can be achieved using SynTS against competing approaches, we performed offline optimization for each approach by varying values of the weight θ (see Equation 4). Figures 4 and 5 show the resulting Pareto curves for energy and execution time for the FMM and Radix benchmarks, respectively, normalized to the Nominal baseline.

We make several observations from these figures. First, we note that both TS approaches have lower best-case execution times than the No-TS baseline; both TS approaches use over-clocking to increase performance. Moreover, SynTS provides high performance at lower energy costs than Per-core TS, 21% lower for FMM and 10% lower for Radix. Second, in its low energy configuration, SynTS is 18% and 8% faster than Per-core TS for FMM and Radix, respectively. In both cases, Per-core TS consumes marginally less ($< 2\%$) energy. The relative savings of SynTS over No-TS are even greater. Finally, although omitted for space constraints, the results for LU are qualitatively similar.

5.2 Online Optimization Results

A critical factor in the success of SynTS online optimization is the fidelity of the error estimates obtained from the sampling phase. Figure 6 shows the actual and estimated error probability functions for an entire barrier interval of the FMM and Radix benchmarks. Here, the length of the sampling phase, N_{sample} , is set to 10% of the total number of instructions in the barrier interval. Observe that in both cases, (1) the estimated error probabilities are close to the actual probabilities, and (2) importantly, the *critical thread from a timing speculation perspective is always identified*. We have observed the same behaviour over other barrier intervals and for the LU benchmark.

For the online experiments, we set N_{sample} to 50K instructions, except for FMM, which has very short barrier intervals. Therefore, for FMM, we set N_{sample} to 10K instructions. In all cases, V_{sample} is set to the nominal chip voltage. For every voltage level in Table 1, each core can choose among six clock periods that are a fraction $r \in [0.64, 1]$ of the nominal clock period. Finally, since our focus here is on addressing heterogeneity in error probability (and the impact of estimating it online), we assume that information on workload heterogeneity (N_i for each thread) is available from offline characterization or using online workload prediction techniques proposed in the literature [12, 6, 2].

Figure 7 plots the execution time, energy and energy-delay product (EDP) of our online implementation of SynTS to competing approaches for the FMM, Radix and LU benchmarks. Results are for a fixed value of θ that weights energy and execution time equally. The results are normalized to the SynTS (offline), allowing us to evaluate the overheads of implementing SynTS online.

Figure 7: Results for three Splash2 benchmarks. All results are normalized to SynTS (offline).

We can make several observations from Figure 7. (1) The overhead of online versus offline SynTS is relatively low—10.3% in EDP on the average over the three benchmarks. These overheads are because of imperfect error probability estimation, and because the sampling phase is executed at sub-optimal voltage and frequency levels to estimate the error probabilities. (2) Notwithstanding the difference, SynTS outperforms the competing approaches for all three benchmarks. Compared to existing timing speculation, online SynTS is better up to 21% in terms of EDP, on an average, for the three benchmarks. The benefits are even greater when compared to No-TS.

6 Conclusion

In this paper, we proposed Synergistic Timing Speculation (SynTS), a novel technique to optimize the energy and execution time of multi-threaded applications executing on multi-cores. SynTS is based on a new empirical observation — heterogeneity in the sensitized delay distributions and thereby the error probabilities under timing speculation across threads. Our empirical evaluation of SynTS illustrates that it improves EDP by up to 21% as compared to per-core timing speculation and up to 28% compared to no timing speculation. As future work, we plan to evaluate a larger set of benchmark applications, and extend our approach to multi-threaded applications that use other synchronization mechanisms, besides barriers.

Acknowledgments

This work was supported in part by National Science Foundation grants (CAREER-1253024, CCF-1318826, CNS-1421022, and CNS-1421068). 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.

7 References

- [1] ANDREW, B. K., ET AL. Recovery-driven design: a power minimization methodology for error-tolerant processor modules. In *DAC* (2010), pp. 825–830.
- [2] BHATTACHARJEE, A., AND MARTONOSI, M. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Comp. Arch. News* (2009), vol. 37, ACM, pp. 290–301.
- [3] BINKERT, N., ET AL. The gem5 simulator. *SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7.
- [4] BOIS, D., AND OTHERS. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 511–522.
- [5] BOWMAN, K. A., ET AL. Energy-Efficient and Metastability-Immune Resilient Circuits for Dynamic Variation Tolerance. *JSSC* 44, 1 (2009), 49–63.
- [6] CAI, Q., ET AL. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *PACT* (2008), ACM, pp. 240–249.
- [7] CARLSON, T. E., ET AL. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *HPCA* (2011), ACM, p. 52.
- [8] DADGOUR, ET AL. Aging-resilient design of pipelined architectures using novel detection and correction circuits. In *DATE* (2010), pp. 244–249.
- [9] DAS, S., ET AL. RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance. *JSSC* 44, 1 (Jan. 2009), 32–48.
- [10] DE KRULF, M., ET AL. A unified model for timing speculation: Evaluating the impact of technology scaling, CMOS design style, and fault recovery mechanism. In *DSN* (2010).
- [11] GHOSH, S., ET AL. A Novel Delay Fault Testing Methodology Using Low-Overhead Built-In Delay Sensor. *TCAD* 25, 12 (2006), 2934–2943.
- [12] LIU, C., ET AL. Exploiting barriers to optimize power consumption of CMPs. In *PDPS* (2005), IEEE, pp. 5a–5a.
- [13] MIHIR, R. C., ET AL. TIMBER: Time borrowing and error relaying for online timing error resilience. In *DATE* (2010), pp. 1554–1559.
- [14] NICHOLAS, J. W., ET AL. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *DSN* (2004), p. 61.
- [15] PARK, S., ET AL. Managing energy-performance tradeoffs for multithreaded applications on multiprocessor architectures. *SIGMETRICS Perform. Eval. Rev.* 35 (2007), 169–180.
- [16] RONG, Y., ET AL. Online clock skew tuning for timing speculation. In *ICCAD* (2011), pp. 442–447.
- [17] WOO, S. C., ET AL. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA-22* (1995), pp. 24–36.
- [18] ZAHRA, L., AND NICOLA, N. In-system and on-the-fly clock tuning mechanism to combat lifetime performance degradation. In *ICCAD* (2011), pp. 434–441.
- [19] ZHAO, W., AND CAO, Y. New Generation of Predictive Technology Model for sub-45nm Early Design Exploration. *IEEE Trans. on Elec. Dvcs.* 53, 11 (2006), 2816–2823.