SYNERGISTIC TIMING SPECULATION FOR MULTI-THREADED PROGRAMS

by

Atif Yasin

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Sanghamitra Roy, Ph.D. Major Professor	Koushik Chakraborty, Ph.D. Committee Member
Vincent Wickwar, Ph.D. Committee Member	Mark R. McLellan, Ph.D. Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2016

ABSTRACT

Synergistic Timing Speculation for Multi-threaded Programs

by

Atif Yasin, Master of Science

Utah State University, 2016

Major Professor: Sanghamitra Roy, Ph.D.
Department: Electrical and Computer Engineering

Timing speculation is a promising approach to increase the processor performance and energy efficiency. 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, which is presented later in the thesis, that these critical path delays are rarely manifested during the 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 any other conventional operation.

While many past works have dealt with timing speculation within a single core, in this work, a new direction is being uncovered — timing speculation for a multi-core processor executing a parallel, multi-threaded application. Through a rigorous cross-layered circuit-architectural analysis, it is observed that during the execution of a multi-threaded program, there is a significant variation in circuit delay characteristics across different threads.

Synergistic Timing Speculation (SynTS) is proposed to exploit this variation (heterogeneity) in path sensitization delays, to jointly optimize the energy and execution time of the many-core processor. 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. The experimental

analysis is presented for three pipe stages, namely, Decode, SimpleALU and ComplexALU, with a reduction in Energy Delay Product by up to 26%, 25% and 7.5% respectively, compared to existing per-core timing speculation scheme. The analysis also embeds a case study for a General Purpose Graphics Processing Unit.

(44 pages)

PUBLIC ABSTRACT

Synergistic Timing Speculation for Multi-threaded Programs

Atif Yasin

Timing speculation is a promising approach to increase the processor performance and energy efficiency. Under timing speculation, an integrated circuit is allowed to operate at a speed faster than the rated speed specified by its vendor. However, doing so might result in an incorrect 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 any other conventional operation.

While many past works have dealt with timing speculation within a single core, in this work, a new direction is being uncovered by exploring timing speculation for a multi-core processor executing a parallel, multi-threaded application. It is observed that during the execution of a multi-threaded program, there is a significant variation in circuit delay characteristics across different threads. Synergistic Timing Speculation (SynTS) is proposed to exploit this variation to jointly optimize the energy and execution time of the many-core processor. The experimental analysis shows significant performance improvement and savings in energy, compared to existing timing speculation schemes.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my adviser, Dr. Sanghamitra Roy, for her insightful advice and patient guidance. Without her motivation and insights this work would have never been complete. Also, I would like to thank Dr. Koushik Chakraborty for his useful critiques of this research, as well as his valuable inputs which helped me tremendously in framing this work. I would like to thank Dr. Siddharth Garg and Jeff from NYU-Poly for their commitment and earnest help in realizing this research work. I would like to thank various student members of Bridge Lab for their constant support, encouragement, as well as making Bridge Lab such a pleasant and rewarding place to work. Specifically, I would like to thank Shamik Saha and Prabal Basu for their technical guidance throughout this research and eagerness to help me out in the times of trouble, and Hu for his constant support. I would like to express my great appreciation to the ECE department and all of the staff members, for offering me this opportunity of Masters research, as well as the financial assistance towards my tuition. I am particularly grateful for the assistance given by Mary Lee Anderson and Tricia Brandenburg, who have helped me through numerous paper works and formatting of the dissertation. I would also like to extend my thanks to Scott Kimber and Hyde for providing technical support. Last but not least, special thanks to my parents and family for their constant support, encouragement and guidance throughout my study.

Atif Yasin

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

GPU	Graphics Processing Unit
CPU	Central Processing Unit
TS	Timing Speculation
EDP	Energy Delay Product
DVS	Dynamic Voltage Scaling
DVFS	Dynamic Voltage and Frequency Scaling
SynTS	Synergistic Timing Speculation
GPGPU	General Purpose Graphics Processing Unit
WCD	Worst Case Design
SIMD	Single Instruction Multiple Data
TSR	Timing Speculation Ratio
CMP	Chip Multi Processor
VALU	Vector ALU

INTRODUCTION

1.1 Worst-Case Design

System reliability is of fundamental importance for ensuring a correct execution of a program. However, sometimes the system execution deviates from its original intended behavior and results in an incorrect output (for the scope of this study, the errors referred to are the timing errors[1]). These errors can manifest because of many reasons, including, process variation and aging etc. In order to ensure that the system is reliable and always results in correct/expected execution of a program, contemporary systems are designed with some slack/guard band to incorporate these rare extreme cases. Since these timing errors occur rarely, for most of the time, system energy is being wasted, which can otherwise be utilized to enhance the performance and/or lower energy consumption. In the quest of achieving the ever higher system performance, Ernst et al. proposed a system design, Razor, to exploit this slack [1].

1.2 Better Than Worst-Case Design

Timing errors occur when the combinational delay of a circuit exceeds the clock period associated to it. If the frequency of the system is increased by eliminating the guard band, it is essentially equivalent to removing the slack designed to cater for such extreme rare cases, leaving the circuit prone to timing errors. In order to detect and correct these timing errors dynamically on-the-go, additional hardware as depicted in Fig. 1.1[2] is added to the system. Error correction is necessary for ensuring the robustness of the pipeline.

The underlying idea proposed by Ernst et al. is to exploit the above mentioned slack by operating an integrated circuit at a higher over-scaled frequency, causing the timing errors to occur frequently [1]. It then detects and corrects the timing errors incurred

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

[2] Ernst, D. et al Razor

instead of avoiding them, ultimately helping in running the system at a higher energy efficiency/performance, while the reliability still intact.

In Figure 1.1, a combinational logic state is being executed at a speculative frequency, resulting in timing errors. The outlined scheme, Razor, uses a shadow latch operated with a delayed clock and in an event of a timing error, the delayed clock latches a different output than the main flip flop output; the Xor gate drives the error bit high for replaying the pipe line and hence correcting the timing error event.

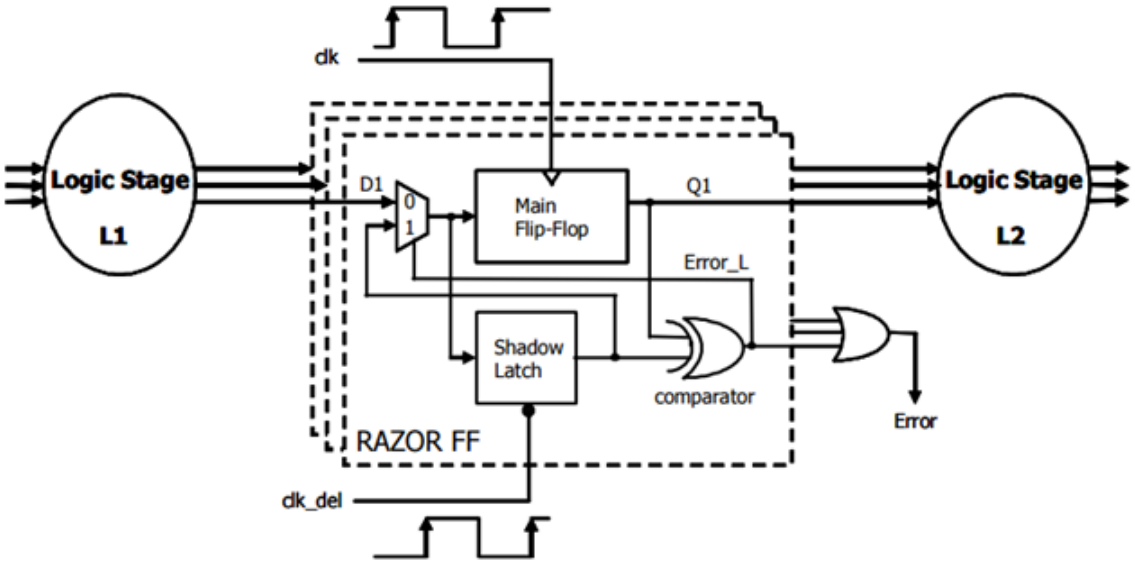


Fig. 1.1: Razor flipflop.

The trade-off for selecting the optimal Timing Speculative frequency is illustrated in Fig. 1.2[3] on the next page. f_r is the rated frequency, specified by the vendor for the system, f_0 is the nominal frequency and f_s is some optimal speculative frequency. Any frequency chosen above this point will result in the degradation of performance, since the penalty caused by flushing and replaying the pipeline will dominate the performance gain achieved by higher frequency, and likewise, any point chosen between f_0 and f_s will be sub-optimal.

[3]Josep Torrellas and Brian Greskamp, "Timing Speculation: Designing Multi-Cores for Single-Thread Performance"

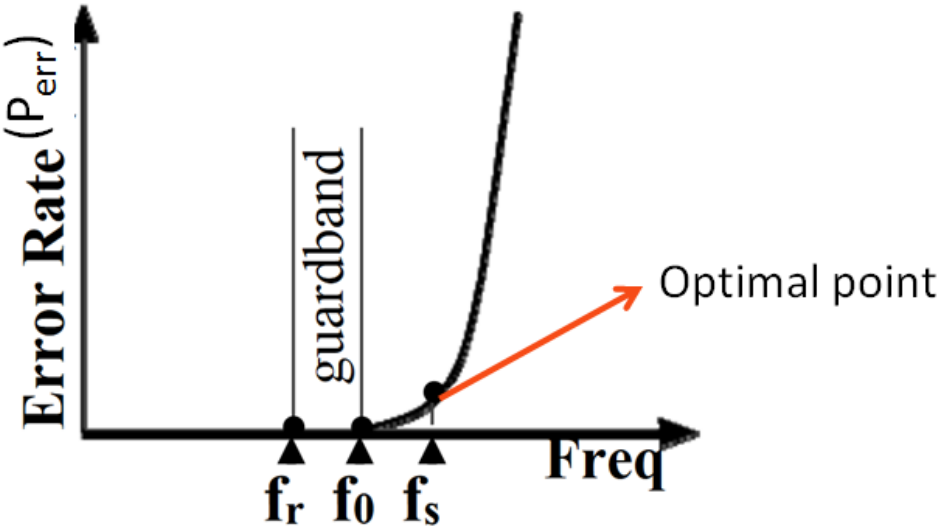


Fig. 1.2: Timing speculation vs. Error probability.

1.3 Timing Speculation: Single-Threaded vs. Multi-Threaded Applications

Having established that for a single-threaded application, an optimal point for a particular speculative frequency can be chosen. That is, for a specific thread, there exists a frequency which yields the highest benefit in the performance gain. However, the scenario changes for the execution of a multi-threaded application. Since there are multiple threads instead of just one thread, optimal point of one thread might not be similar to that of the other threads, that is, the timing error probability might not be same for all threads. Choosing an optimal point independently for all cores ignores a fact that execution of a program in a multi-threaded system is dependent on other threads. This, hence, requires a formal architectural analysis of an execution of a multi-threaded application to allow us to analyze the effect of frequency up-scaling in all cores holistically, so that the maximum benefit can be attained out of timing speculation.

1.4 Multi-Threaded Application and Barrier Synchronization

In order to better understand the inner mechanics of a multi-threaded application and the underlying dependency among threads, in this section, a multi-threaded workload execution is dissected. Fig. 1.3 shows a brief snapshot of a multi-threaded workload. Since,

the execution of a multi-threaded application is dependent on all of the threads, they need to synchronize at certain points, so that their execution goes hand in hand [2]. One typical implementation of rendezvous points is barrier synchronization; a thread arriving at the barrier interval can only pass that synchronization point when all threads have arrived at that point. This type of synchronization makes sure that no thread is left behind and all the dependency of data between threads is maintained properly [3]. However, it is possible for the threads not to arrive at the barrier points simultaneously [4]. As it is shown later in this paper using empirical data, this is what happens in the multi-threaded workload as shown in Fig. 1.4. There exists "faster" threads and "slower" threads and due to this difference, faster threads have to wait at barrier points so that the slower threads can catch up and all threads can move forward across the barrier point. This idle waiting for the slower critical threads to arrive at barrier can be exploited to improve performance and/or save the energy.

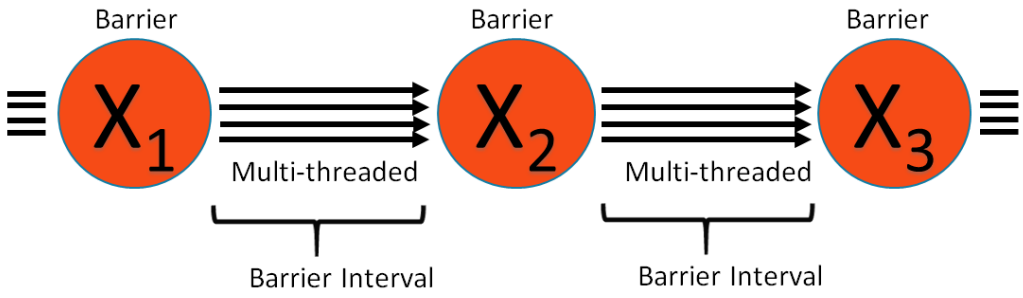


Fig. 1.3: Multi-threaded workload execution.

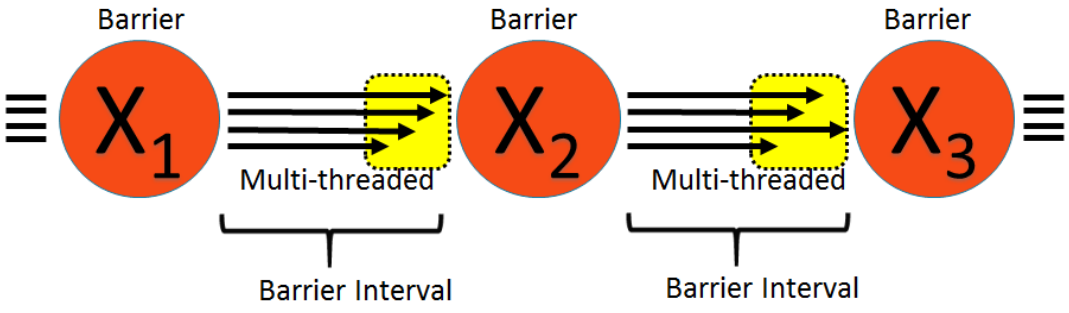


Fig. 1.4: Threads arriving at barrier at different times.

RELATED WORK

Previous work most relevant to this 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. RazorII based dynamic voltage scaling allows elimination of safety margins and operation at the point of first failure of the processor. This technique deals with the variation induced delay-errors, specifically the PVT (Process,Voltage and Temperature) and SER (Soft error rate) [6]. Other techniques allow elimination of clock frequency safety margins from dynamic voltage supply while correcting the timing errors [17, 18].

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. These techniques mask timing errors by borrowing time from successive pipeline stages [19–21]. Several other works advocate dynamic clock skewing [22, 23], 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 a multi-core processor.

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 [8, 15, 24]. None of these works address timing speculation for multi-threaded applications and are oblivious of the fact that the error probability functions of a multi-threaded workload are heterogeneous in nature or exploit

timing speculation criticality, which this work does for the first time.

MOTIVATIONAL EXAMPLE

As mentioned in the previous section, timing speculation for a multi-threaded application depends on all the threads executing in a multi-core processor. Fig. 3.5 shows an example of a timing speculation critical thread in the Radix benchmark from the SPLASH-2 benchmarks suite (See Section 4.3 for detailed methodology.) In this example, Thread 0 consistently has the highest error probability with decreasing clock period, about 4x greater than the thread with the lowest error probability.

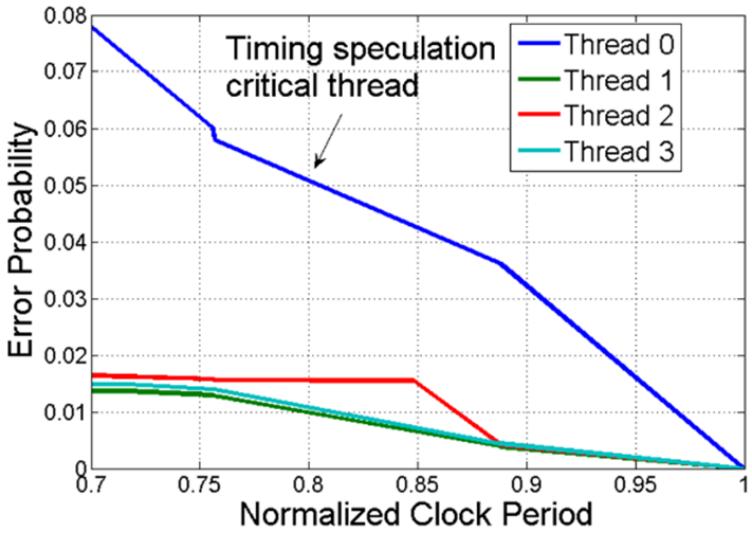
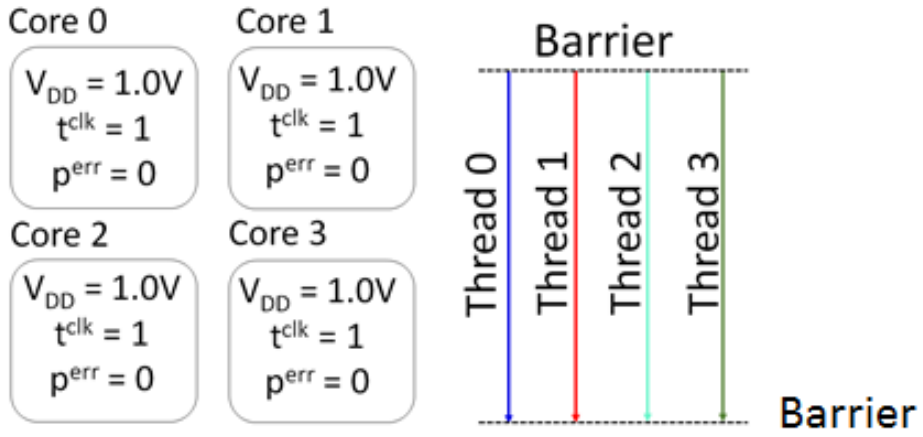


Fig. 3.5: Timing error probability with a normalized clock period for one barrier interval in the Radix benchmark.

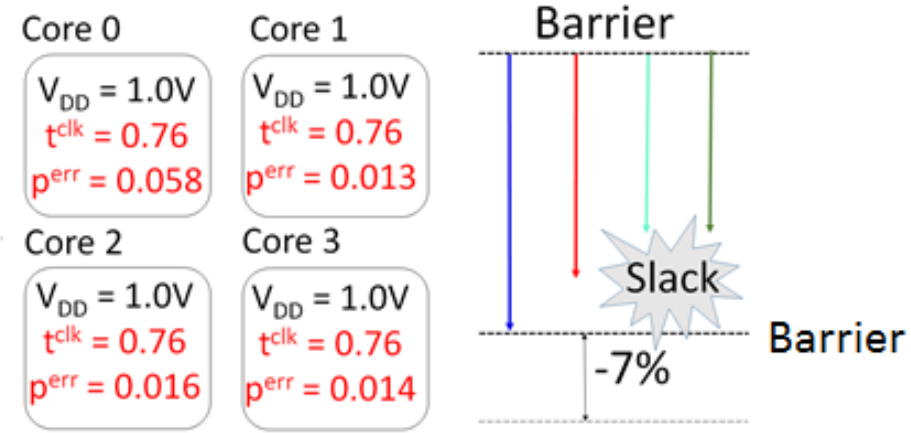
3.1 Synergistic Timing Speculation Approach

To exploit this intriguing thread-level heterogeneity in timing errors characteristics, Synergistic Timing Speculation for multi-threaded workloads is proposed in this thesis by incorporating timing speculation criticality. The approach is referred to as SynTS.

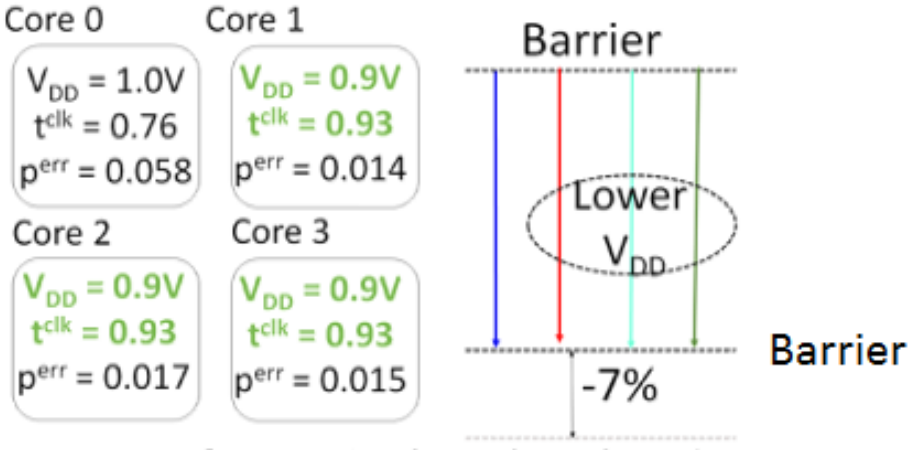
Figure 3.6 explains the opportunities of SynTS to substantially improve the system energy efficiency by exploiting heterogeneity in thread error probabilities.



(a) Nominal baseline - Same voltage and frequency for all cores. All threads reach the barrier at the same time.



(b) Step-1 - Frequency up-scaling and gaining performance benefit. A slack created because of heterogeneity in thread error probabilities.



(c) Step 2 - Voltage downscaling and adjusting frequency to save on energy efficiency

Fig. 3.6: Overview of the SynTS approach. The data here is generated based on the error probability curve in Figure 3.5. More details about the algorithm and experimental parameters are given in Section 3.3 and Section 4.3, respectively.

The example initially presents a scenario where all four threads run at the nominal voltage and frequency without timing speculation (Each core has only 1 thread). An assumption is made that all 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 latencies.

In step 1, timing speculation tends 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 3.6, reducing the clock period by 24%, reduces execution time of Thread 0 by 7%, observed empirically. The execution times of Thread 1, 2 and 3 are reduced even further, since they have lower error probabilities. Any further increase in clock frequency hurts the performance, since it causes an increase in timing errors and its associated recovery overheads in Thread 0, nullifying the positive effects of higher frequency. At the end of Step 1, although all the threads reach the barrier sooner than in the nominal case, Thread 0 is now critical, i.e., it reaches the barrier last, while the other three threads have some slack.

In Step 2, the slack that has been created is leveraged 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 is answered in this thesis: How can the optimal voltage, frequency, and as a result the error probability be synergistically determined (for each thread/core) so as to jointly optimize for execution time and energy consumption?

3.2 Exploring the GPGPUs

In the above mentioned analysis, a case study for a 4 core Alpha Processor is presented

and have motivated that there is a distinct heterogeneity in error probabilities since multiple threads exist in a multi-threaded application. GPGPUs are an excellent candidate for timing speculation as well, since GPGPUs architecture resemble to that of CMPs and contain thousands of threads and multiple streaming cores, providing an extensive parallelism [5]. This motivates to perform thread level path sensitization delay analysis for GPGPUs as well.

3.3 Thesis Contributions

This thesis makes the following contributions and first in the domain of exploring Timing Speculation in the multi-threaded applications for multi-core systems including GPGPUs.

- Using an elaborate cross-layer methodology, empirical evidence of heterogeneity in error probabilities of threads under timing speculation for SPLASH-2 benchmarks applications is provided. SynTS approach is implemented that is aware of this heterogeneity and synergistically performs timing for all the threads to optimize the performance and energy.

- A mathematical formulation for SynTS is implemented as a discrete optimization problem, along with a polynomial time algorithm that optimally solves this problem. Subsequently, a practical online implementation of SynTS based on error probability sampling and polynomial time algorithm is proposed.

- Using a rigorous circuit-architectural simulation environment, a significant improvement in energy-efficiency of CMPs – up to 26%, 25% and 7.5% reduction in the energy delay product (EDP) for Decode, SimpleALU, and ComplexALU respectively when compared to existing timing speculation schemes.

- Exploring the architecture of Radeon HD 7970 GPGPU for the presence of heterogeneity in error probabilities among 16 Streaming Multiprocessors in inter-SIMD unit containing thousands of parallel threads.

SynTS DESIGN

In this section, the design of SynTS for CMPs is discussed in three steps. First, a mathematical model for timing speculation on multi-core processors is discussed (Section 4.2). Then, ideal offline implementation of SynTS is discussed (Section 4.1) and finally a practical online version is presented (Section 4.3).

4.1 System Model

In this thesis, a multi-core processor consisting of M homogeneous cores is considered 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 [6].

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., $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_Q\}$. For every voltage level $V \in \mathcal{V}$, there is a *nominal clock period* $t^{nom}(V)$ at which 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; $t_i^{clk} = r_i t^{nom}(V_i)$. The symbol r_i is referred to as *timing speculation ratio* (TSR), and assumed that it is picked from one of S discrete levels including 1. That is $r_i \in \mathcal{R}$, where $\mathcal{R} = \{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_S = 1\}$. Note that the TSR implicitly corresponds to the discrete clock frequency at which the 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 is shown in Figure 3.5. As observed before, the error probability func-

tion can *vary from one thread to another*, i.e., it is thread specific. With these preliminaries, system performance and energy consumption can be modeled based on the one proposed by Kruijf et al. for processors with fine-grained error recovery mechanisms like Razor [7] . In particular, the seconds per instruction (SPI)[4] of thread i can be written as:

$$SPI_i = t_i^{clk}\left(p_i^{err}C^{penalty} + CPI_i^{base}\right) \tag{4.1}$$

where $C^{penalty}$ is the error recovery penalty (5 cycles for the Razor processor [7]) and CPI_i^{base} is the baseline clock per instruction of thread i in the absence of errors.

The focus of this thesis is on parallel applications that use barrier synchronization. Without the loss of generality, the application execution time for a single barrier interval is measured; the total execution time can be obtained by summing over all barrier intervals. 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 [8]:

$$t_{exec} = \max_{i\in[1,M]} N_i t_i^{clk}\left(p_i^{err}C^{penalty} + CPI_i^{base}\right) \tag{4.2}$$

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

$$en_i = \alpha V_i^2 N_i\left(p_i^{err}C^{penalty} + CPI^{base_i}\right) \tag{4.3}$$

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. Although the model does not currently account for leakage power, it can be easily extended to do so. Furthermore, although the focus of this thesis is for exploring the *energy* versus execution time trade-offs, the proposed approach can be generalized to address power consumption as well.

4.2 Offline Optimization

[4] Inverse of instruction per second (IPS).

This sections describes the proposed synergistic timing speculation methodology for CMPs. To begin with, it is optimistically assumed that the workload characteristics, that is, each thread's error probability function, $err(r_i)$, is known in *advance*. In Section 4.3, the above mentioned assumption will be relaxed and detail an *online* policy that estimates the error probability function for each thread on the fly.

The 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**,

$$\min_{V_i, r_i} \sum_{i \in [1,M]} en_i + \theta t_{exec} \tag{4.4}$$

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.

4.2.1 MILP Formulation

SynTS-OPT is a discrete, non-linear optimization problem. It is first reduced to a mixed integer linear programming (MILP) problem, which is referred to as **SynTS-MILP**. To do so, binary variables x_{ijk} are introduced, 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:

$$min_{x,en,p^{err},t^{clk},t_{exec}} \sum_{i \in [1,M], j \in [1,Q], k \in [1,S]} x_{ijk} en_{ijk} + \theta t_{exec}, \tag{4.5}$$

subject to:

$$t_{exec} \geq N_i t_i^{clk} \left(p_i^{err} C^{penalty} + CPI_i^{base} \right) \quad \forall i \in [1, M], \tag{4.6}$$

$$t_i^{clk} = \sum_{j \in [1,Q], k \in [1,S]} x_{ijk} \mathcal{R}_k t^{nom}(\mathcal{V}_j) \qquad \forall i \in [1, M], \tag{4.7}$$

$$p_i^{err} = \sum_{j \in [1,Q], k \in [1,S]} x_{ijk} err_i(\mathcal{R}_k) \qquad \forall i \in [1, M], \tag{4.8}$$

$$en_{ijk} = N_i \left(p_i^{err} C^{penalty} + CPI_i^{base} \right) \alpha \mathcal{V}_j^2 x_{ijk} \quad \forall i \in [1, M], \tag{4.9}$$

and

$$\sum_{j \in [1,Q], k \in [1,S]} x_{ijk} = 1 \qquad \forall i \in [1, M]. \tag{4.10}$$

Equation 4.6 constrains execution time of the barrier to be larger than that of each thread, while Equations 4.7, 4.8 and 4.9 compute the clock period, error probability and energy, respectively, in terms of the x_{ijk} variables. Equation 4.10 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 this thesis.

The intuition behind this algorithm, **SynTS-Poly**, is as follows. It 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, all combinations of voltage and timing speculation ratios are tried; for each such combination, thread's execution time is obtained ($\mathcal{O}(QS)$ iterations). Then, for every other thread, *lowest energy* configuration that allows it to finish before or with the critical thread is searched ($\mathcal{O}(MQS)$ 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, the algorithm returns 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 4.2.1. *Algorithm 1 is guaranteed to return an optimal solution for* ***SynTS-OPT****.*

Proof. The optimality of the algorithm can be proved based on the fact that there exist at least one critical thread, among all other threads, having the longest execution time. Since

the algorithm iteratively assume each thread as a critical thread, there exists a case where this assumption is true. Performing SynTS under that correct assumption, all possible combinations of voltage and timing speculation ratios will be searched. Therefore, the algorithm will return the optimal result. Note that it depends on the fact that the non-critical threads only impact the energy component of the cost function. □

Algorithm 1: SynTS-Poly optimization procedure

Algorithm `SynTS-Poly`$(\mathcal{V}, \mathcal{R}, t^{nom}, err, C^{penalty}, CPI^{base})$
  for $i \in [1, M]$ **do**
    for $j \in [1, Q]$ **do**
      for $k \in [1, S]$ **do**
        $p^{err} \leftarrow err_i(\mathcal{R}_k)$;
        $t^{clk} \leftarrow t^{nom}(\mathcal{V}_j)\mathcal{R}_k$;
        $t_{exec} \leftarrow N_i t^{clk}\left(p^{err}C^{penalty} + CPI_i^{base}\right)$;
        $en \leftarrow \alpha\mathcal{V}_j^2 N_i\left(p^{err}C^{penalty} + CPI_i^{base}\right)$;
        for $l \in [1, M] \wedge l \neq i$ **do**
          $en \leftarrow en + minEnergy(l, t_{exec})$;
        end
        $cost_{ijk} \leftarrow en + \theta t_{exec}$;
      end
    end
  end
  return $\arg\min_{ijk} cost_{ijk}$;
Procedure `minEnergy`$(i,\ t_{exec})$
  for $j \in [1, Q]$ **do**
    for $k \in [1, S]$ **do**
      $en_{jk} \leftarrow \infty$;
      $p^{err} \leftarrow err_i(\mathcal{R}_k)$;
      if $N_i t^{nom}(\mathcal{V}_j)\mathcal{R}_k\left(p^{err}C^{penalty} + CPI_i^{base}\right) \leq t_{exec}$ **then**
        $en_{jk} \leftarrow \alpha\mathcal{V}_j^2 N_i\left(p^{err}C^{penalty} + CPI_i^{base}\right)$;
      end
    end
  end
  return $\min_{jk} en_{jk}$;

4.3 Online Optimization

In practice, the error probability versus the clock period data is *not* available in advance, and must be estimated on-the-fly. Here, an online sampling based approach is proposed 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}. Figure 4.7 shows an overview of this procedure.

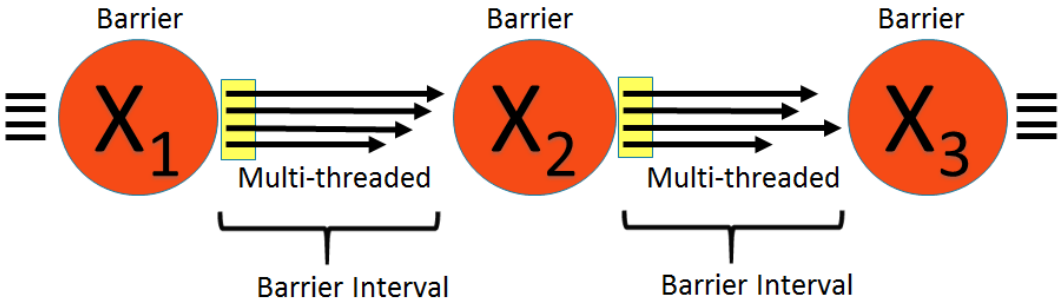


Fig. 4.7: Sampling phase at the start of each barrier interval.

At the end of the sampling phase, an estimate of the error probability function is obtained, err_i, for each thread i. This estimated error probability function is referred to as $\tilde{err}_i$. 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 $\tilde{err}_i(\frac{t_i^{clk}}{t^{nom}(V)})$.

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 the 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 the execution time overhead of sampling. Section 5.5 discusses how values for these parameters are set.

EVALUATION METHODOLOGY

5.1 Cross-Layer Methodology

In this section, cross-layer methodology has been described which motivates for the need of a synergistic timing speculation strategy. Fig. 5.8 gives an overview of the methodology used.

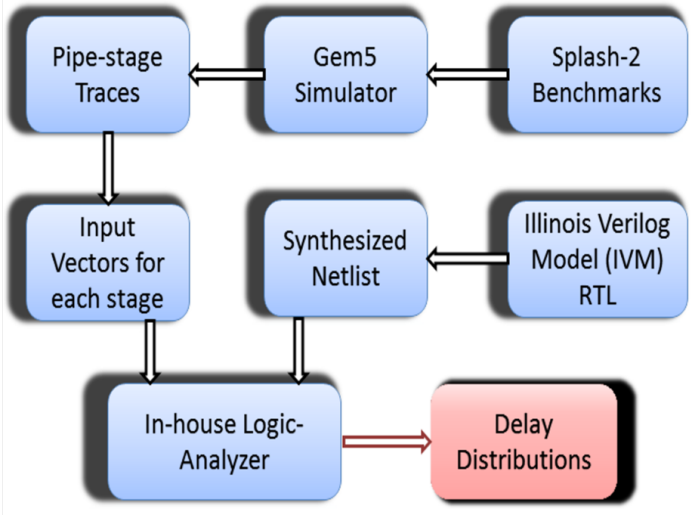


Fig. 5.8: Cross-layer methodology to profile delay characteristics.

5.2 Architectural Simulator

For performing the architectural simulations, a cycle-accurate Gem5 simulator [9] is used to model a 4-core Alpha processor. Several Splash2-benchmarks representing a range of real world applications [10] are executed on the simulator to 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 the program execution. Each benchmark is executed for 3 barrier intervals, or to its completion, whichever comes first. For the GPGPUs analysis, a cycle-accurate Multi2sim

4.2 simulator [11] is used to model a Radeon HD 7970 GPGPU processor. Several Splash-2 [10] and Parsec 2.0 [12] benchmarks are executed for the analysis.

5.3 Timing Error Modeling

Using Synopsys Design Compiler, Illinois Verilog Model [13] of the Alpha processor is synthesized to obtain a gate-level netlist for each pipe stage; and MIAOW rtl [5] for Radeon HD 7970 processor to obtain gate-level netlist for a SIMD unit within the Compute Unit. Next, by feeding cycle-by-cycle input vectors for each stage to its structural RTL netlist, sensitized paths for each instruction is recorded. In this work, the analysis is performed for decode, SimpleALU and ComplexALU pipe stages of a CMP. The propagation delays of gates on the sensitized paths are obtained from HSPICE simulations using the Predictive Technology Model (PTM) for the 22nm node [14]. Finally, to model the impact of voltage scaling on the propagation delay, HSPICE is used to simulate 22 nm ring oscillators and record the clock period versus voltage, as shown in Table 5.1. Based on this cross-layer methodology, traces of propagation delays for each instruction are recorded.

Table 5.1: 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

5.4 Benchmarks-CMPs

The above mentioned methodology is used to characterize the error probability functions of 10 SPLASH-2 benchmarks— FMM, Radix, LU-contig, LU-ncontig, FFT, Water-sp, Barnes, Raytrace, Cholesky and Ocean. Of these, FFT, Ocean and Water-sp have homogeneous error probabilities for all threads, for which conventional timing speculation and proposed approach (SynTS) would work just as well. In fact, the FFT error probabilities are high and do not permit any timing speculation. Hence, results for the remaining 7 benchmarks from the Splash-2 suite are reported in Section 5.5.

5.5 Benchmarks-GPGPUs

Again, the above mentioned methodology is used to characterize several benchmarks including BlackScholes, EigenValue, MatrixMult, FFT, BinarySearch, Raytrace, StreamCluster, Swaptions and X264. For this particular architecture of GPGPU, all the multi-threaded applications demonstrated homogeneous error probabilities among all 16 streaming cores in a particular SIMD unit.

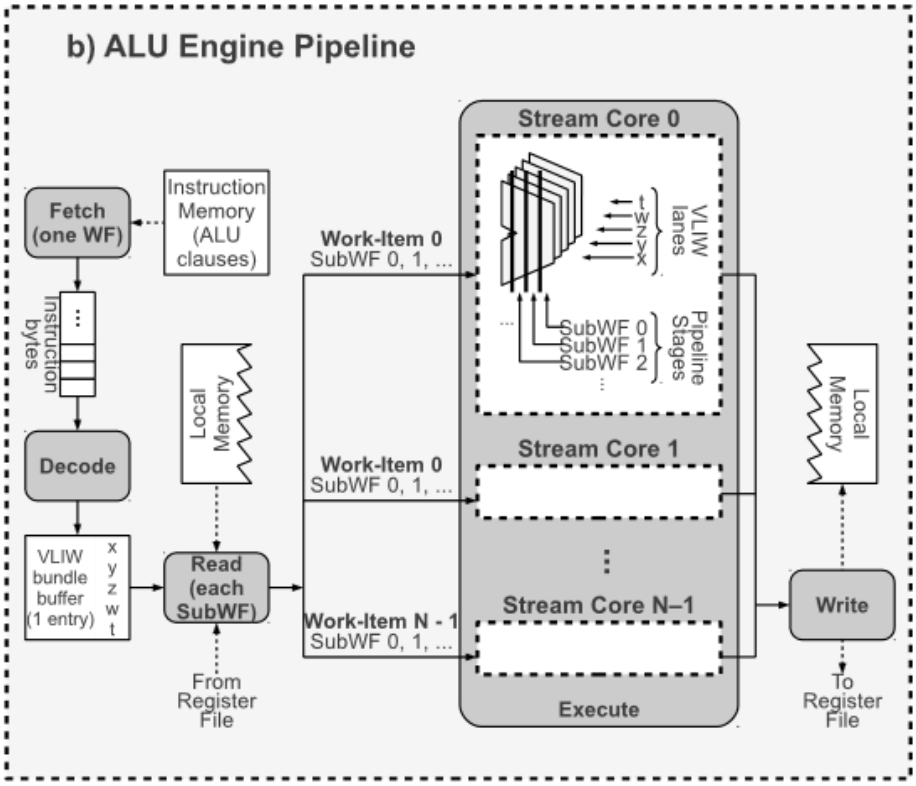
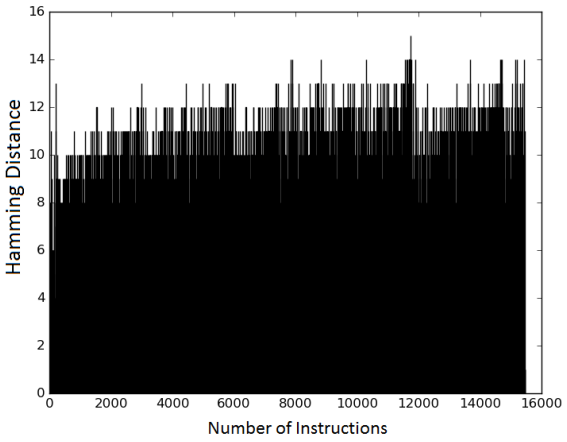


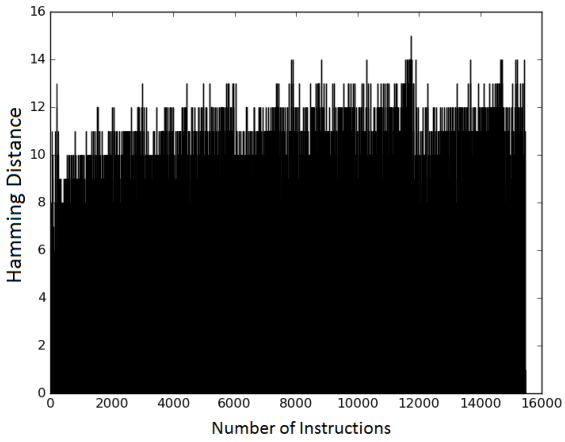
Fig. 5.9: Streaming Cores and Vector ALUs in HD Radeon 7970.

In order to better understand this homogeneity in error probabilities, output obtained from the 16 SMs/Vector ALUs (Figure 5.9[5] shows the VALU(VLIW lanes)) is analyzed to plot the hamming distance bar graphs. For each Work-Item (set of instructions), cycle-by-cycle instruction inputs to VALU are extracted to analyze its output. Figure 5.10 represent the hamming distance graphs for different VALU's showing almost a similar trend in the output.

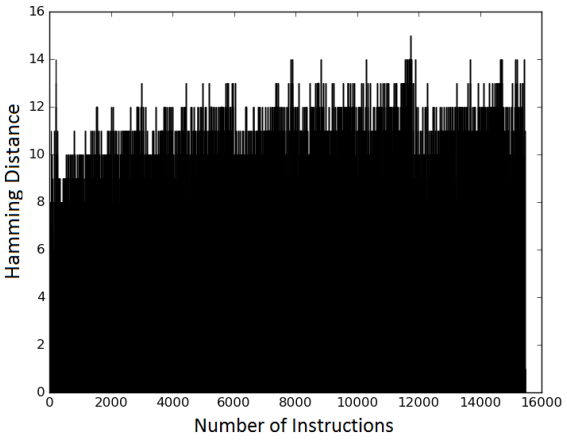
[5]Ubal, Rafael, et al. "Multi2Sim: a simulation framework for CPU-GPU computing



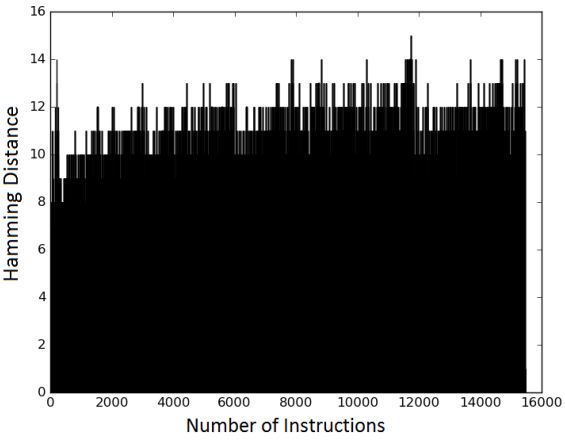
(a) Vector ALU 0



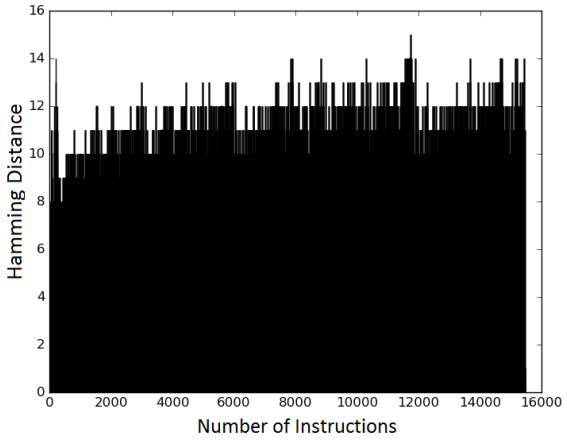
(b) Vector ALU 1



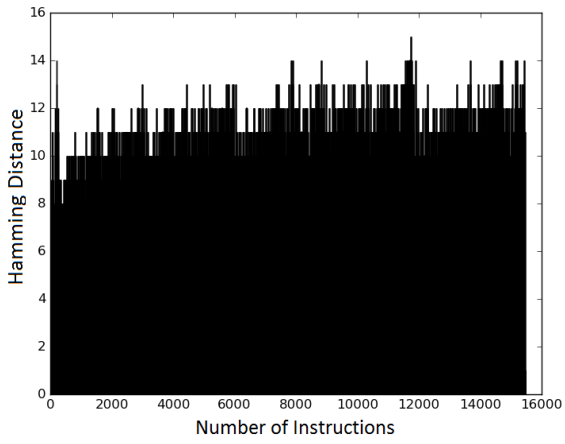
(c) Vector ALU 2



(d) Vector ALU 3



(e) Vector ALU 4



(f) Vector ALU 5

Fig. 5.10: Hamming distance bar graphs for the output of 6 vector ALUs. The graphs for rest of the VALUs are qualitatively similar.

The data is shown for 6 VALUs and for 16k instructions, however, the trend is almost similar for remaining 10 VALUs, evaluated over 100k instructions. Similar hamming distance means that the output characteristics of different VALUs are similar and trends in the path sensitization delays are also similar. This evaluates to the homogeneity in error probabilities. Hence, the per-core timing speculation will work just fine for this particular architecture and workload.

EXPERIMENTAL RESULTS

Since the error probabilities of GPGPU were homogeneous, the results of synergistic timing speculation analysis is presented for the CMPs only. The proposed SynTS approach is compared to several other 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 4.4, 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 [15].
- 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 4.4. Per-core TS serves as a best possible bound for single-core timing speculation techniques like Razor [6], since it has offline access to the error probability functions for each core.

The results for the offline version of SynTS are presented followed with the results for a practical online implementation of SynTS.

6.1 Offline Optimization Results

The above mentioned schemes are initially compared to SynTS in an 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 the quantification of the best results that can be obtained from SynTS.

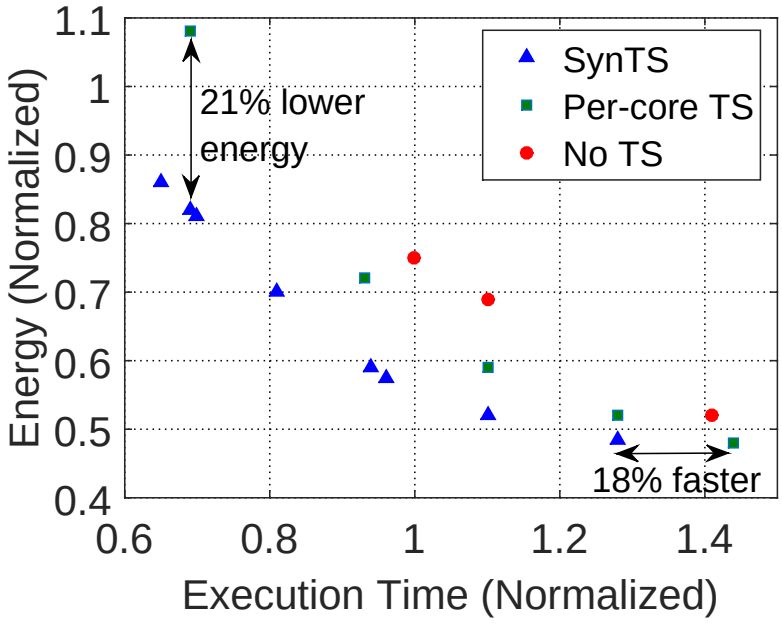


Fig. 6.11: Energy versus execution of FMM for offline versions of the SynTS, Per-core TS and No-TS approaches, normalized to the Nominal baseline. Each point corresponds to a different value of weight θ from Equation 4.4 - (SimpleAlu).

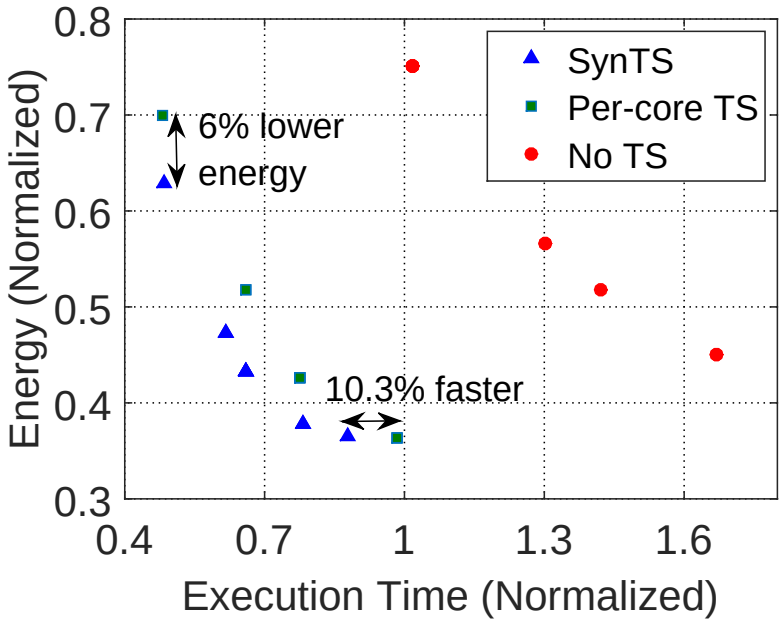


Fig. 6.12: Same data as in Figure 6.11 for the Cholesky benchmark - (SimpleAlu).

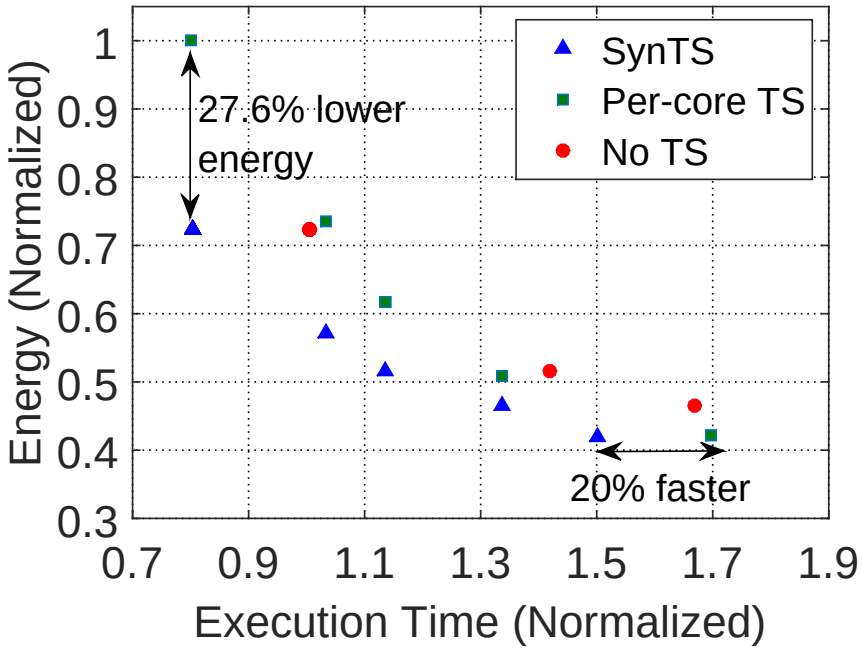


Fig. 6.13: Same data as in Figure 6.11 for the Cholesky benchmark - (Decode).

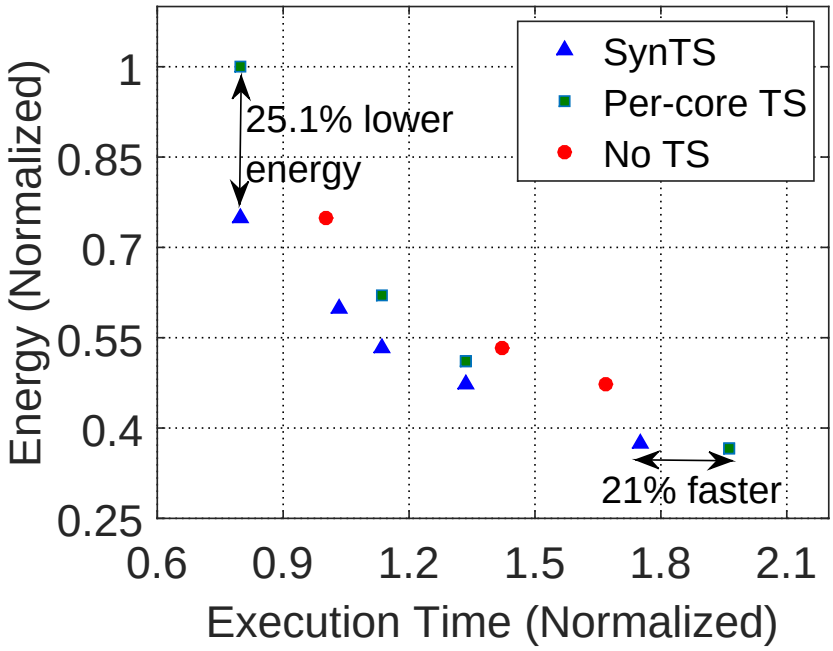


Fig. 6.14: Same data as in Figure 6.11 for the Raytrace benchmark - (Decode).

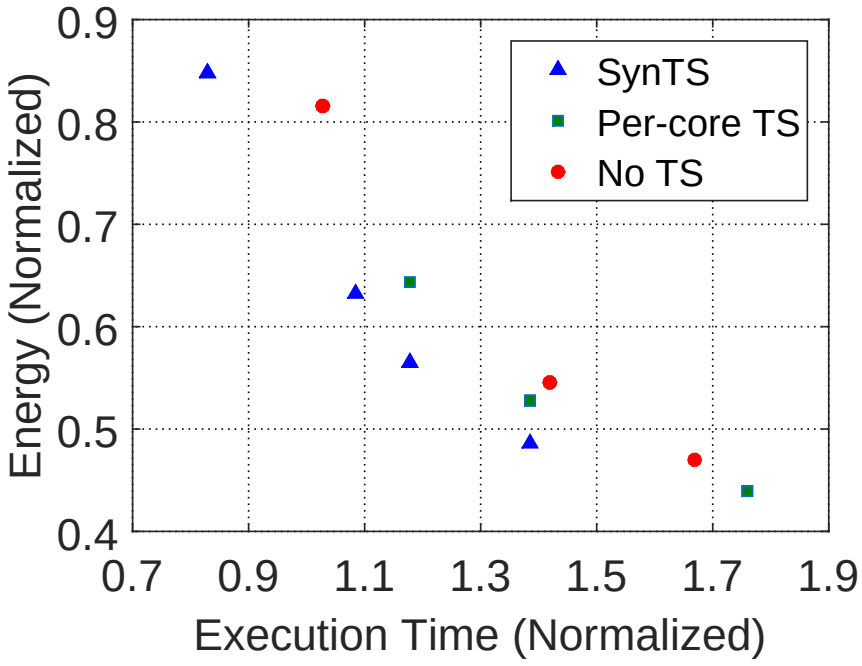


Fig. 6.15: Same data as in Figure 6.11 for the Cholesky benchmark - (ComplexAlu).

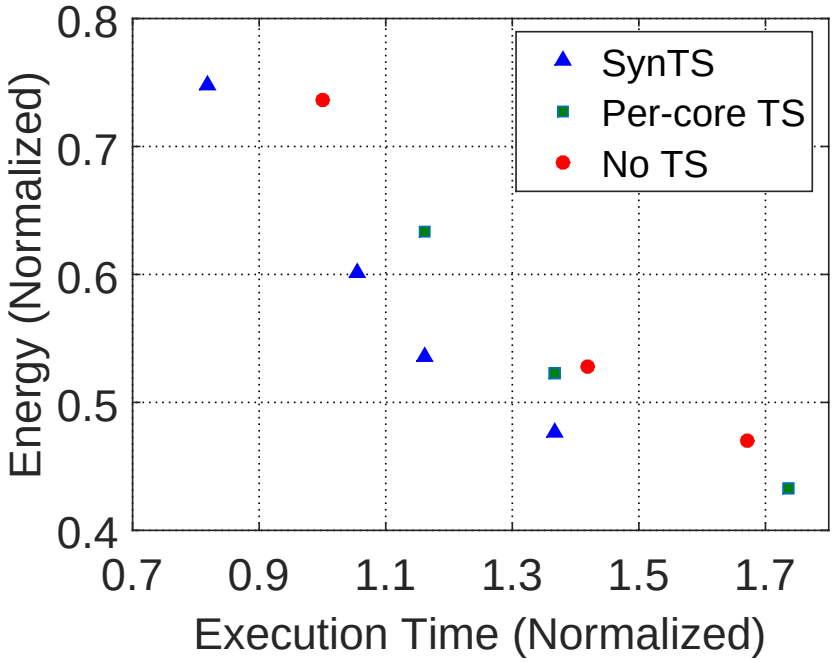


Fig. 6.16: Same data as in Figure 6.11 for the Raytrace benchmark - (ComplexAlu).

To compare the energy vs. execution time trade-offs that can be achieved using SynTS against competing approaches, offline optimization for each approach is performed by varying values of the weight θ (see Equation 4.4). Figure 6.11 to 6.16 show the resulting Pareto curves for energy and execution time for Decode, SimpleAlu and ComplexAlu for different benchmarks, normalized to the Nominal baseline. Several observations are made from these figures. First, 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 6% lower for Radix, respectively, for SimpleAlu; 27.6% lower for Cholesky and 25.1% lower for Raytrace for Decode. ComplexAlu also shows lower energy as compared to other approaches, however, direct comparison can not be drawn since Per-core TS and No TS do not converge close to SynTS. Second, in its low energy configuration, SynTS is 18% and 10.3% faster than Per-core TS for FMM and Radix ,respectively, for SimpleAlu; 20% and 21% faster for cholesky and raytrace, respectively, for Decode. In all of these 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 other benchmarks are qualitatively similar for all three pipe-stages analyzed.

6.2 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 6.17 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*. Similar behavior is observed for all barrier intervals and for the rest of the benchmarks over all three stages.

For the online experiments, N_{sample} is set to 50K instructions, except for benchmarks which have very short barrier intervals, interval size is adjusted accordingly. One such

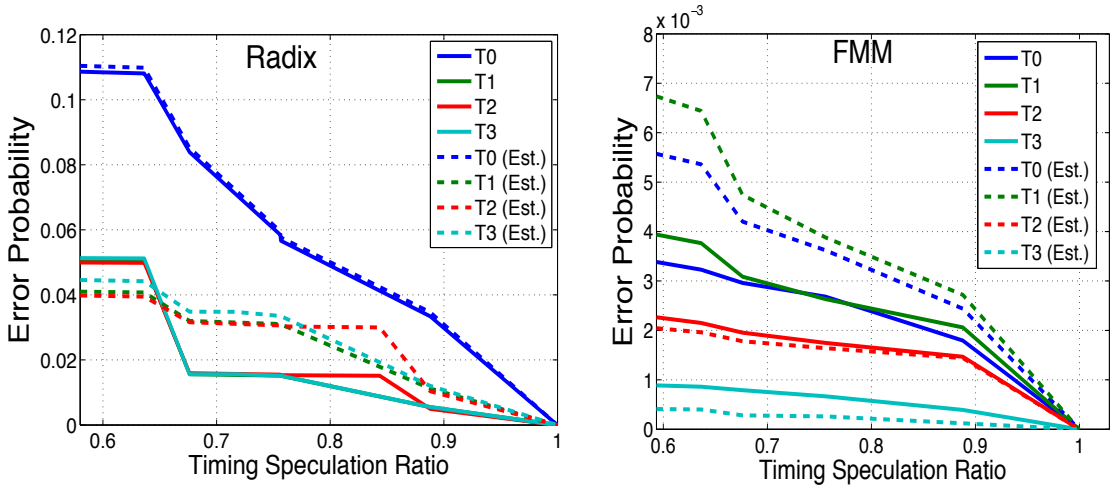
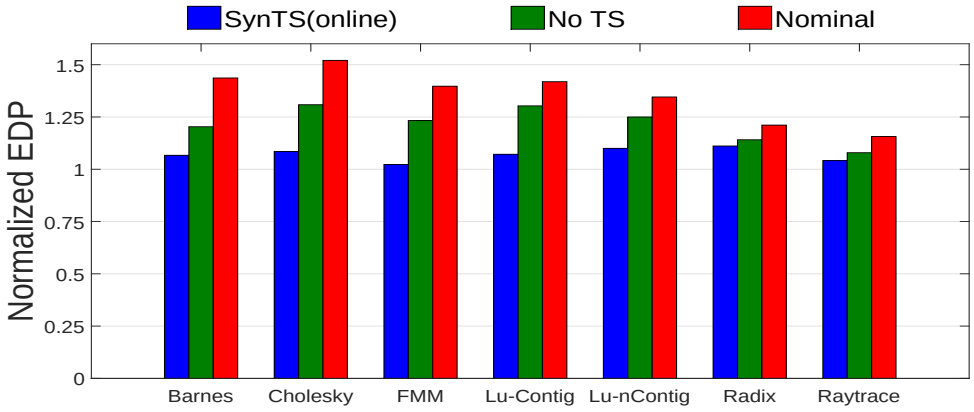


Fig. 6.17: Actual and online estimated error probability versus timing speculation ratio for one barrier interval in the Radix and FMM benchmark.

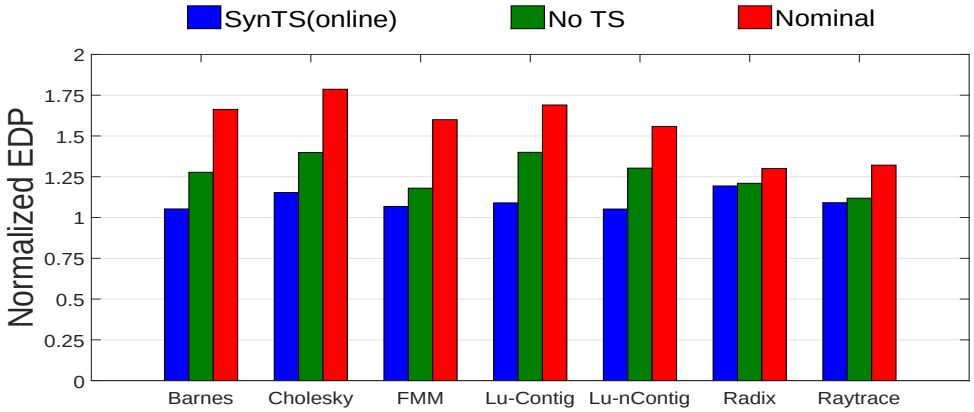
example is for the benchmark FMM, where the N_{sample} is set to 10K instructions. In all cases, V_{sample} is set to the nominal chip voltage. For every voltage level in Table 5.1, each core can choose among six clock periods that are a fraction $r \in [0.64, 1]$ of the nominal clock period. Finally, since the focus here is on addressing heterogeneity in error probability (and the impact of estimating it online), it is assumed that the information on workload heterogeneity (N_i for each thread) is available from offline characterization or using online workload prediction techniques proposed in the literature [8, 15, 16].

Figure 6.18 plots the energy-delay product (EDP) of proposed online implementation of SynTS to competing approaches for various benchmarks across three pipe stages. 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.

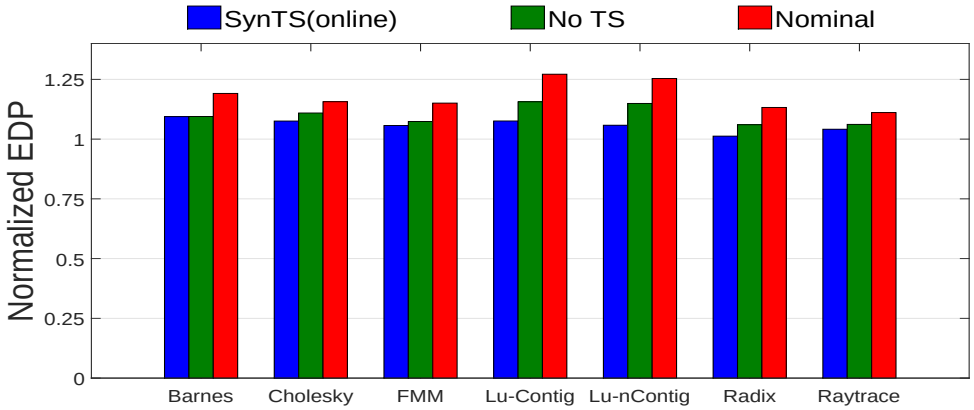
Several observations can be drawn from Figure 6.18. (1) The overhead of online versus offline SynTS is relatively low—10.3% in EDP on the average over the seven benchmarks. These overheads are because of imperfect error probability estimation, and because the sampling phase is executed at sub-optimal voltage and frequency levels to estimate the



(a) Decode



(b) Simple-ALU



(c) Complex-ALU

Fig. 6.18: Energy Delay Product of seven Splash2 benchmarks for Decode, Simple-Alu and Complex-Alu - Normalized to its respective SynTS (offline).

error probabilities. (2) Notwithstanding the difference, SynTS outperforms the competing approaches for all seven mentioned benchmarks and for all three pipe stages. Compared to existing timing speculation, online SynTS is better up to 25% in terms of EDP. The benefits are even greater when compared to No-TS.

6.3 Optimization Overhead of SynTS-Online

As mentioned earlier, 10% of the instructions are sampled at the start of each barrier interval. The primary power overhead in SynTS is attributed to the sampling process of finding the timing speculation critical thread within each barrier interval. The power overhead of proposed scheme is calculated by synthesizing the pipe stages from the IVM 1.0 using the Synopsys Design Compiler with a 45nm FreePDK library.

The overhead relative to the core power and area is estimated after adding all the hardware enhancements of the SynTS. Overall, the power overhead is around 3.41% for SynTS (online) relative to the core power when sampled for barrier intervals over one benchmark, on average. The area overhead of SynTS (online) is even smaller, at 2.7% when compared to the core.

CONCLUSION

In this thesis, a novel technique, Synergistic Timing Speculation (SynTS), has been proposed to optimize the energy and execution time of multi-threaded applications executing on multi-core processors. SynTS is based on a new empirical observation — heterogeneity in the sensitized delay distributions and thereby the error probabilities under timing speculation across different threads. Observing the sensitized-delay variance across different threads, SynTS adjusts the operating-frequency and supply voltage for each thread to make the thread-barrier synchronization more energy-efficient. One of the key objective of this study was to improve the system energy efficiency and/or performance. Our empirical evaluation of SynTS illustrates that it improves EDP by up to 25% as compared to per-core timing speculation and up to 55% compared to no timing speculation.

As future work, this approach can be extended to multi-threaded applications that use other synchronization mechanisms, besides barriers for CMPs. For GPGPUs, other architectures can be analyzed, however, lack of resources (open-source GPGPU rtl and corresponding architectural simulator) restrict the analysis for the time being.

REFERENCES

[1] D. Ernst *et al.*, "Razor: A low-power pipeline based on circuit-level timings speculation," in *Microarchitecture. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on. IEEE*, 2003.

[2] T. E. Carlson *et al.*, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC-11*, 2011.

[3] T. E. Jeremiassen and S. J. Eggers, "Static analysis of barrier synchronization in explicitly parallel programs," in *IFIP PACT*. Citeseer, 1994, pp. 171–180.

[4] J. Li, J. F. Martinez, and M. C. Huang, "The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors," in *Software, IEE Proceedings-*. IEEE, 2004, pp. 14–23.

[5] R. Balasubramanian *et al.*, "Miaow-an open source rtl implementation of a gpgpu," in *Low-Power and High-Speed Chips (COOL CHIPS XVIII), 2015 IEEE Symposium in*. IEEE, 2015, pp. 1–3.

[6] S. Das *et al.*, "RazorII: In situ error detection and correction for PVT and SER tolerance," *JSSC*, vol. 44, no. 1, pp. 32–48, Jan. 2009.

[7] M. De Kruijf *et al.*, "A unified model for timing speculation: Evaluating the impact of technology scaling, cmos design style, and fault recovery mechanism," in *DSN*, 2010.

[8] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *Comp. Arch. News*, vol. 37, no. 3. ACM, 2009, pp. 290–301.

[9] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[10] S. C. Woo *et al.*, "The splash-2 programs: Characterization and methodological considerations," in *ISCA-22*, 1995, pp. 24–36.

[11] R. Ubal *et al.*, "Multi2sim: a simulation framework for cpu-gpu computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 335–344.

[12] C. Bienia *et al.*, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[13] J. W. Nicholas *et al.*, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *DSN*, 2004, p. 61.

[14] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45nm early design exploration," *IEEE Trans. on Elec. Dvcs.*, vol. 53, no. 11, 2006.

[15] C. Liu *et al.*, "Exploiting barriers to optimize power consumption of cmps," in *PDPS*. IEEE, 2005, pp. 5a–5a.

[16] Q. Cai *et al.*, "Meeting points: using thread criticality to adapt multicore hardware to parallel regions," in *PACT*. ACM, 2008, pp. 240–249.

[17] K. A. Bowman *et al.*, "Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance," *JSSC*, vol. 44, no. 1, pp. 49–63, 2009.

[18] B. K. Andrew *et al.*, "Recovery-driven design: a power minimization methodology for error-tolerant processor modules," in *DAC*, 2010, pp. 825–830.

[19] Dadgour *et al.*, "Aging-resilient design of pipelined architectures using novel detection and correction circuits," in *DATE*, 2010, pp. 244–249.

[20] R. C. Mihir *et al.*, "Timber: Time borrowing and error relaying for online timing error resilience," in *DATE*, 2010, pp. 1554–1559.

[21] S. Ghosh *et al.*, "A novel delay fault testing methodology using low-overhead built-in delay sensor," *TCAD*, vol. 25, no. 12, pp. 2934–2943, 2006.

[22] Y. Rong *et al.*, "Online clock skew tuning for timing speculation," in *ICCAD*, 2011, pp. 442–447.

[23] L. Zahra and N. Nicola, "In-system and on-the-fly clock tuning mechanism to combat lifetime performance degradation," in *ICCAD*, 2011, pp. 434–441.

[24] D. Bois and others., "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *ACM SIGARCH Computer Architecture News*, 2013, pp. 511–522.