Reducing Performance Impact of Process Variation For Data Caches

Ismail KADAYIF, Kadir TUNCER

Department of Computer Engineering, Canakkale Onsekiz Mart University, Canakkale, Turkey
kadayif@comu.edu.tr; kadirtuncer@comu.edu.tr

Abstract

In concurrent with finer-granular process technologies, it is becoming extremely difficult to keep critical physical device parameters within desired bounds, including channel length, gate oxide thickness, and dopant ion concentration. Variations in these parameters can lead to dramatic variations in access latencies in Static Random Access Memory (SRAM) devices: Different lines of the same cache may have different access latencies. A simple solution to this problem is to adopt the worst-case latency paradigm. While this egalitarian cache management is simple, it may introduce significant performance overhead for data cache accesses. To overcome varying access latencies across different data cache lines, we employ a small table storing the access latencies of cache lines. This table is accessed during data cache access to give a hint to the hardware about how long to wait for data to become available.

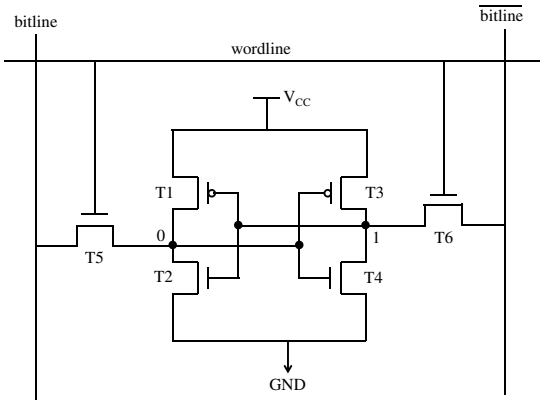

Figure 1. Six-transistor SRAM cell storing one bit.

1. Introduction

While, over the last three decades, scaling of Complementary Metal Oxide Semiconductor (CMOS) devices has improved the performance of computer systems dramatically, keeping transistor quality within desired bounds has been becoming a challenging problem. This problem is referred to as process variation and can be described as the deviation from intended or designed target values of a circuit parameter of concern, such as channel length or width, gate oxide thickness, and random placement of dopants in a channel. It can lead to significant variability in chip performance, power consumption, and stability [1, 2, 3]. Such variations may occur not only across identically designed neighboring devices [4] - intra-die variations - but also across different identically designed chips [5] - inter-die variations.

In this study, we focus on intra-die variations in cache memories, because they have become more prevalent as we go into deep sub-micron silicon technologies. Especially, the random placement of dopants, as a result of difficulties in handling finer process technologies, can lead to a threshold voltage mismatch among transistors in the same hardware component. SRAM structures are quite prone to random placement of errors due two major reason: First, they constitute a significant portion of die area (for instance, for Alpha 21264 and Strong ARM, 30% and 60% of the die areas, respectively, are devoted to cache structures [6]) and their share increases with next technology generation; Second, SRAM structures are typically designed with minimum-sized transistors for density reasons [7]. As a result, different cache lines of the same cache may have different performance and energy behavior.

Although threshold voltage mismatch can cause read/write stability failures [8, 9], in this study, we focus on access time failures (performance effects) of threshold voltage fluctuations of neighborhood transistors in SRAMs. Let us explain how threshold fluctuations of devices in a SRAM cell can affect the cell's access time. Figure 1 depicts a six transistor cell, which is commonly employed in SRAM designs. The cache access time strongly depends on the cell access time, which is defined as the interval required to see a specific voltage difference (say, $\Delta_{MIN} \approx 0\ 1\ _{CC}$) between two bitlines. For a read operation, for example, both bitlines are first precharged, and then the wordline is enabled (it is set to high). If the cell stores 0, as in the figure, the bitline on the left side will discharge through transistors T5 and T2. Any variation in the threshold voltages ($_t$) of these transistors due to a process parameter variation can extend the discharging interval, causing access time failure if the delay is larger than the maximum tolerable limit. If a cache line accommodates a cell with access time failure, accesses to this cache line must be delayed to give the failed cell some extra time to achieve the required voltage difference on the corresponding bitlines; this in turn results in delay violations in the cache structures. More specifically, for such SRAM structures, the access latency will not be uniform; i.e., the access latency varies across different cache lines. An easy way to handle varying access latencies in a cache structure is to adopt the worst-case access latency paradigm in the design: We delay the access latency of regular cache lines - cache lines which are not affected by process variation - to that of the cache line which are the most severely affected. While this simple assumption makes the design simple, it may lead a significant performance penalty in regular cache line accesses. This performance penalty increases as we move to the finer process technologies [10, 11].

In this study, we try to mitigate the performance overheads on data cache accesses due to process variation in modern processors. We use a small table, which is called as *latency table*,

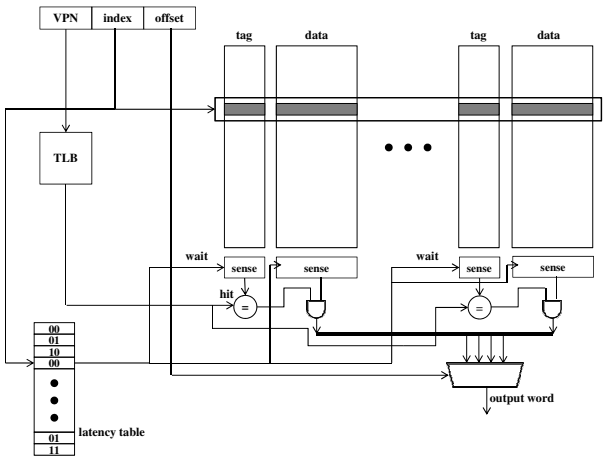

Figure 2. Accessing the data cache experiencing process variation. The latency information of the second pipe stage for each data cache line is encoded in the latency table, which is accessed at the same time when index bits of the data cache are decoded. The timing of the second pipe stage of the data cache is adjusted according to the value read out of the latency table.

to store the access latencies of each cache set. The latency table is managed by the hardware and is accessed at the same time when the data cache is accessed. The value read out of this table gives the hardware a hint about how long to wait for the data to become available.

The rest of this paper is structured as follows. In the next section, we try to explain how the latency table is organized and accessed in tandem with the data cache access. In Section 3, we explain how we can minimize the number of faulty cache sets. The experimental setup is given in Section 4. Experimental results are explained in Section 5. Finally, our concluding remarks are given in Section 6.

2. Data Cache Access Under Process Variation

The data cache access under process variation is depicted in Figure 2. In our study, we focus on pipelined caches, since they have becoming more prevalent due to their boosting performance. We consider a 3-stage pipelined cache model as suggested in [12]. According to this model, set address is decoded in the first stage. Wordline driving, bitline precharging and monitoring the voltage difference between a pair of bitlines by sense amplifiers take place in the second stage. Driving the output multiplexors and the selected data out of the cache are carried out in the last stage. Unlike the second stage, the first and the last stages may be further divided into substages. Since the bitline signals are weak, not digital, latching is possible only after the voltage difference of analog bitlines is converted into the digital by sense amplifiers, making the second stage indivisible [12]. Process variation may affect all of the these three stages; however, the second stage is the most critical since it cannot be divided further to lessen the performance effect of the variations. Therefore, in our study we take only the variations that affect the second pipe stage into consideration. The latency table encodes, for each cache set, the second pipe stage's latency.

Our latency table works on set basis: Since at compilation time, in general, only the set (not the cache line) in which the data reside can be determined (in our study we consider set associative instruction caches, not direct mapped caches), we consider the latencies at the set granularity (assuming that each set has a single latency, which is defined as the largest latency among all its lines).

To obtain the performance characteristics of the cache lines, we can employ the March test [9], which was originally proposed to test memory components' functionality, and involves a sequence of operations performed on different locations in the memory. With this test, cache line *i* is characterized either as regular (not affected by process variation), $= 0$; or as faulty (affected by process variation), $= 1$.

As can be seen from Figure 2, the data cache and the latency table are accessed simultaneously. Our basic assumption is that the latency table access finishes until the first pipe stage of the data cache is completed. This holds since the latency table is very small. For example, 64 KB, 4 way set associative data cache with 64 byte cache lines, the number of cache sets is 256. And if we encode 4 different access latencies, each table entry holds 2 bits ($lo\ 4 = 2$), requiring a latency table with the size of $2\ x\ 256 = 512$ bits. The latency value read out of the latency cache gives a hint to sense amplifiers to how long wait before sending their output signals to multiplexors, which exercise in the last pipe stage. Thus, the length of the second pipe stage varies, depending on the corresponding devices in the second pipe stage experiences process variation as well as the severity of the variation.

3. Minimizing the Number of Faulty Cache Sets

Our scheme relies on storing the latency information of the second pipe stage of cache access in the latency table. Since the access latencies stored in the table is set basis, rather than line basis, it is possible that a set can include regular lines as well as faulty lines. If this is the case, an access to a regular line in the faulty set will be delayed, resulting in performance loss. Thus, it is very important to minimize the number of imperfect sets.

To do so, we use our technique called *line reshuffling* [13], which is similar to *block rearrangement techniques* proposed by Mutyam and Narayanan [14]. They considered block rearrangement either between a pair of two adjacent cache sets or among all cache sets. On the other hand, in our technique, we perform line reshuffling among a specific number of cache sets, which can be implemented by a programmable address decoder. The control inputs of the pass transistors driving the word lines are programmed in such a way that any set is allowed to include th cache lines with the same access latency as much as possible. For simplification, we confine reshuffling to way boundaries, that is, reshuffling is allowed only among the cache lines within the same way. We use *reshuffling degree*, the term to refer to the number of address bits involving in reshuffling. If *r* and *s* denote reshuffling degree and the number of sets, reshuffling is done among 2^r cache lines in consecutive sets belonging to the same way. These lines constitute a reshuffling group and their sets can be expressed as follows,

$$2^r \ \leq \ < 2^r(\ +1)\ where\, 0 \leq \ < \ 2^r$$

More details of our technique can be found in [13].

Table 1. Major processor configuration parameters and their values used in our experiments.

Processor Core	
Functional Units	4 Integer ALUs, 2 Integer mult./divide, 4 FP add, 2 FP multiply, 2 FP divide/sqrt
RUU size	256 instructions
LSQ size	64 instructions
Fetch/Decode/Issue/Commit width	4 instructions/cycle
Fetch queue size	8 instructions
Cache and Memory Hierarchy	
L1 instruction cache	64KB, 4-way (LRU), 64 byte blocks, pipelined with 3-cycle latency
L1 data cache	64KB, 4-way (LRU), 64 byte blocks, pipelined with 3-cycle latency
L2 cache	8MB unified, 8-way (LRU), 128 byte blocks, 12-cycle latency
Memory	300-cycle latency
Page size	8K
Branch Prediction	
Branch predictor	Combined, Bimodal 4K table, 2-level 2K table, 8-bit history, 4K chooser
Branch target buffer (BTB)	4K-entry, 4-way
Return-address stack	32

Table 2. Benchmarks used in our experiments and their important characteristics.

Benchmark Name	Number of Execution Cycles	Number of Data Cache Accesses
bwaves	411954109	135420769
mgrid	284257419	154870565
applu	301706421	160920926
gamess	188126703	141902084
gzip	238701269	98261019
vpr	504901292	94231020
gcc	248621430	74891701
mcf	701231064	108931619

4. Experimental Setup

We evaluated our technique by modifying SimpleScalar 3.0 [15], a tool set that simulates application programs on a range of processors and systems using a fast execution-driven simulation, and outputs execution statistics, such as the dynamic number of accesses to components in the memory hierarchy as well as execution cycles. In this study, the *sim-outorder* component of SimpleScalar has been modified to simulate the integration of our technique into an Alpha-like platform. The major simulation parameters of our target processor are listed in Table 1. We used some codes from the SPEC2000 suite (mgrid, applu, gzip, and vpr) as well as some codes from the SPEC2006 suite (bwaves, gamess, gcc, and mcf) [18] in our experiments. Since the simulation takes a long time to run the benchmarks, we considered, to completion, we used SimPoint [16] to generate simulation points. For each benchmark, we fast forwarded a specific number of instructions, as suggested by Sherwood et al. [17], and then simulated the next 500 million instructions on predetermined simulation points. Table 2 gives the number of execution cycles and the number of data cache accesses of these benchmarks under the configuration parameters listed in Table 1.

5. Experimental Results

Before delving into the details of experimental results, we want to explain the schemes whose experimental results are under consideration.

- *Perfect Scheme*: This reflects the case where there is no cache line experiencing process variation, and each data cache access takes 3 cycles to complete. This setup corresponds to ideal case. The results given in Table 2 belong to this scheme.
- *Delayed Scheme*: This corresponds to worst-case paradigm; accessing any cache line takes an amount of time which is equall to the access time of the slowest cache line. Although it is a very simple solution to the problem of process variation, as explained later, it introduces huge performance degradation.
- *Oracle Scheme*: It is useful from theoretical perspective, rather practical perspective, and can provide a basis to measure the the effectiveness of our scheme. In this scheme, we have a hypothetical predictor, which can foretell for each cache access whether the corresponding cache line is affected by process variation. If the line is regular, the cache access is completed in 3 cycles; otherwise the cache access takes an amount of time depending on the severity of the variations.
- *Latency Table (LT) Scheme*: This is our scheme and works with the latency table. The basic premise of this scheme is that, since the size of the latency table is very small compared to the data cache, an access to this table is completed until the first pipe stage of the cache access is finished. The latency value read out of the latency cache gives a hint to sense amplifiers in the second stage regarding how long to wait before sending their output signals to multiplexors, which exercise in the last pipe stage. Thus, the length of the second pipe stage varies, depending on the corresponding devices in the second pipe stage experiences process variation as well as on the severity of the variation.

Here, we need to mention that the latency table itself may be subject to process variation. However, because of the small wordline capacitance of the few cells used in the table, even with process variation, an access to this table is finished before the first pipe stage of the data cache access completes.

In the experiments involving process variation, we take only the variations into consideration that affect the second pipe stage, delaying the stage by 1, 2, or 3 cycles with an equal probability. More specifically, the regular cache line accesses complete in 3 cycles while variation affected cache accesses complete in either 4,5, or 6 cycles. Since we need to encode four different access latency for the second pipe stage, we need 2 bits for each entry in the latency table. Moreover, we carry out experiments for which 20% and 40% of cache lines are subjected to process variation.

In our experiments, Monte-Carlo simulations were done to model process variation. We have determined 5 different instruction cache setups, each one having randomly distributed variation-affected lines. The performance simulations for each benchmark were run 5 times, each one with a different cache setup. Each performance result was calculated by taking the average of the performance results from these 5 setups.

Figure 3 and Figure 4 present the results of the Delayed and LT schemes for the cases where 20% and 40% of cache blocks are assumed to be affected by variations. In both graphs, there

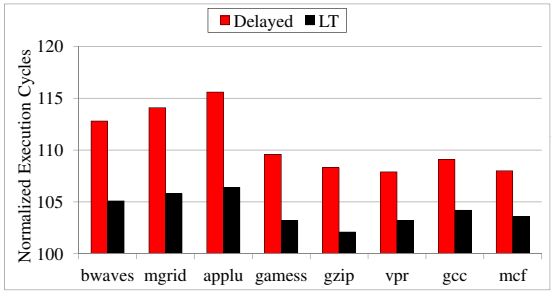

Figure 3. Normalized execution cycles for the case where 20% of cache blocks are assumed to be affected by variations.

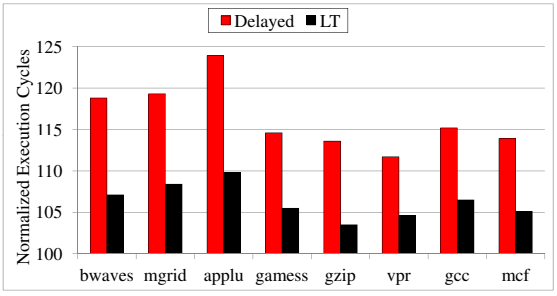

Figure 4. Normalized execution cycles for the case where 40% of cache blocks are assumed to be affected by variations.

are two bars for each benchmark: the first bar indicates the performance of the Delayed scheme while the second bar depicts the performance of the LT scheme. The performance results are given as normalized values with respect to values of the Perfect scheme, the ideal case (the case where the data cache is immune to process variation). From both figures, we can see that the performance degradation become more severe as the percentage of faulty cache block increases. The severity of performance loss introduced by the process variation is more prevalent for the scientific application programs (bwaves, mgrid, applu, and gamess), since they are, in general, more data intensive compared to the integer benchmarks considered (gzip, vpr, gcc, and mcf). The average performance values of the Delayed and LT schemes are around 10.7% and 4.2% when the faulty cache lines are 20%. When we increase to the percentage of the faulty cache lines to 40%, the The average performance values of the Delayed and LT schemes become 16.4% and 6.3%. The two major implications of these results are: First, the worst-case design (the Delayed scheme) can introduce a quite large performance penalty, which is not tolerable for high performance microprocessors; Second, the LT scheme can alleviate the performance loss due to process variation quite effectively.

We indicate the performance comparison of the LT scheme to the Oracle scheme in Figure 5. The left hand side corresponds to the case where 20% of cache lines are assumed to be affected by variations, while the right hand side corresponds to the case where 40% of cache lines are assumed to be affected by variations. The Oracle scheme is useful from theoretical perspective, rather practical perspective, and can provide us a clue regarding to what extend the performance loss introduced by the variations can be overcome. The performance loss of the Oracle scheme is 3.5% and 5.4% when the percentage of the faulty cache lines are 20% and 40%. Considering the performance values of the Oracle scheme as lower bounds, we can easily say that the LT is quite effective to lessen the consequences of process variation from the performance point of view.

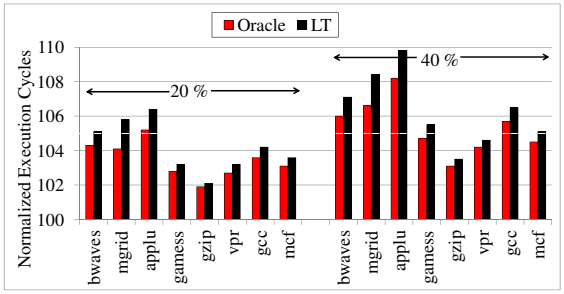

Figure 5. Performance comparison of the LT scheme to the Oracle scheme. The left hand side corresponds to the case where 20% of cache blocks are assumed to be affected by variations, while the right hand side corresponds to the case where 40% of cache blocks are assumed to be affected by variations.

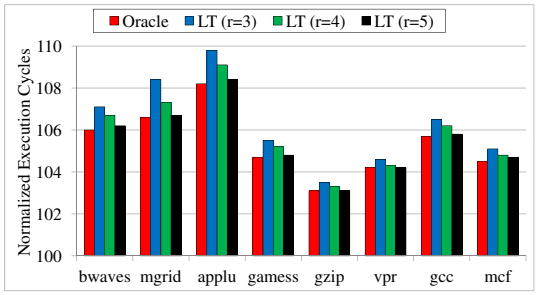

Figure 6. Performance comparison of the LT scheme to the Oracle scheme based on reshuffling degree. The reshuffling degree is varied from 3 to 4 to 5. As reshuffling degree increases, we have a greater opportunity to place cache lines with the same latency characteristics into the same set, hence approximating the LT scheme's performance to that of the Oracle scheme. 40% of the cache lines are assumed to be affected by process variation.

Till now, for the experiments involving process variation, we implicity assume that the reshuffling degree is 3. This means that cache lines belonging to the same way in 8 consecutive sets are considered for reshuffling process. To evaluate the performance of the LT scheme further, we vary the reshuffling degree from 3 to 4 to 5. This means that 8, 16, or 32 adjacent sets are considered as a group and the reshuffling process is implemented within the group. For each reshuffling degree, we have fixed the percentage of the faulty cache blocks to 40%. The experimental results are shown in Figure 6. The larger the reshuffling degree, the greater opportunity to place the cache lines with the same latency characteristic for the second pipe stage into the same set, hence approximating the LT scheme's performance to that of the Oracle scheme. This can be easily observed from the figure. As mentioned before, the average performance of the Oracle scheme across all application programs tested are about 5.4% away from that of the Perfect scheme, where the cache lines are assumed to be immune to process variation.

On the other hand, the average performance values of the LT scheme are approximately 6.3%, 5.9% and 5.5% away from that of the Perfect scheme. This indicates that while the LT scheme is very effective in reducing the performance ramifications of process variation on the data cache with small reshuffling degree, the reshuffling with degree 5 brings the performance of the LT scheme very close to that of the Oracle scheme. Due two reasons, we haven't carried out experiments for the reshuffling degree larger than 5. First, beyond 5 there is a very little opportunity to improve the data cache's performance further, as suggested by our experiments. Second, the area overhead due to reshuffling can be prohibitive beyond 5.

6. Conclusion

In concurrent with delving into deep sub-micron process technology, it is becoming increasingly difficult to control transistor quality within desired bounds. As a result, process variation is emerging as an important problem in system design for SRAM-based memory components like on-chip caches. Process variation may cause fluctuations in access latencies as well as increased power consumption of identically designed components. While working with the worst-case latency assumption makes things a lot simpler, our analysis of the data cache clearly indicate that the performance hit resulting from this scheme is intolerable, especially for data intensive applications. In our study, we have proposed an approach to lessen the performance ramifications of process variation for pipelined L1 data caches with 3 stages. Process variation may affect all of the these three stages; but the second stage is the most critical since it cannot be divided further to lessen the performance effect of the variations. Therefore, in our study we take only the variations into consideration that affect the second pipe stage. In our approach, we employ a small table storing the second pipe stage's latency. To determine performance characteristics of the cache lines, we can employ the March test. Our basic premise is that the time required to access this table isn't longer than the time it takes to complete the first pipe stage of the data cache access. This table is accessed during data cache access to give a hint to the hardware about how long to wait for data to become available. From our experimental results, we see that our scheme is quite effective in alleviating the performance loss introduced by process variation. As future work, we will focus on mitigating the performance ramifications of process variation without using a latency table. Especially, we will try to annotate the cache access latency incurred by load/store instructions within themselves to give the circuitry a hint about how long to wait for the data targeted by the next load/store to become available. To do so, we will investigate some compiler techniques to determine the target cache sets of load/store instructions.

7. References

[1] S. Borkar, T. Karnik, S.Narendra, J. Tschanz, A. Keshavarzi, and V.De, "Parameter Variations and Impact on Circuits and Microarchitecture", *Proc. of the 40th Annual Design Automation Conference*, 2003, pp. 338–342.

[2] H. Chang and S. S. Sapatnekar, "Full-chip Analysis of Leakage Power Under Process Variations, Including Spatial Correlations", *Proc. of the 42nd Annual Design Automation Conference*, 2005, pp. 523–528.

[3] X. Fu, T. Li and J. A. B. Fortes, "Soft Error Vulnerability Aware Process Variation Mitigation", *Proc. of the International Symposium on High Performance Computer Architecture*, 2009, pp. 93–104.

[4] A. Agarwal, D. Blaauw and V. Zolotov, "Statistical timing analysis for intra-die process variations with spatial correlations", *Proc. of the International Conference on Computer Aided Design*, 2002, pp. 900–907.

[5] S. R. Nassif, "Modeling and analysis of manufacturing variations", *Proc. of the IEEE Conference on Custom Integrated Circuits*, 2001, pp. 223–228.

[6] S. Manne, A. Klauser and D. Grunwald, "Pipeline gating: speculation control for energy reduction", *Proc. of the International Symposium on Computer Architecture*, 1998, pp. 132–141.

[7] A. Papanikolaou, F. Lobmaier, H. Wang, M. Miranda, and F. Catthoor, "A system-level methodology for fully compensating process variability impact of memory organizations in periodic applicatios", *Proc. of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2005, pp. 117–122.

[8] A. Agarwal, B. C. Paul, S. Mukhopadhyay, and K. Roy, "Procecess variation in embedded memories: failure analysis and variation aware architecture", *IEEE Journal of Solid-State Circuits*, vol. 40, no. 9, pp: 1804–1814, 2005.

[9] Q. Chen, H. Mahmoodi, S. Bhunia, and K. Roy, "Modeling and testing of SRAM for new failure mechanisms due to process variations in nanoscale CMOS", *Proc. of the 23rd IEEE VLSI Test Symposium*, 2005, pp. 292–297.

[10] K. Bowman, S. G. Duvall and J. D. Meindl, "Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration", *IEEE Journal of Solid-State Circuits*, vol. 37, no. 2, pp:183–190, 2002.

[11] P. S. Zuchowski, P. A. Habitz, J. D. Hayes and J. H. Oppold, "Process and environmental variation impacts on ASIC timing", *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, 2005, pp. 336–342.

[12] Z. Chishti and T. N. Vijaykumar, "Wire delay is not a problem for SMT (in the near future)", *Proc. of the International Symposium on Computer Architecture*, 2004, pp. 40–51.

[13] I. Kadayıf, M. Turkcan, S. Kiziltepe, and O. Ozturk, "Hardware/software approaches for reducing the process variation impact on instruction fetches", *ACM Transactions on Design Automation of Electronic Systems*, in print, 2013.

[14] M. Mutyam and V. Narayanan, "Working with process varation aware caches", *Proc. of the Design, Automation and Test in Europe Conference*, 2007, pp. 1–6.

[15] SimpleScalar toolset, http://www.simplescalar.com.

[16] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: faster and more flexible program analysis", *Journal of Instruction-Level Parallelism*, vol. 7, pp:1–25, 2005.

[17] T. Sherwood, E. Perelman and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications", *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2001, pp. 3–14.

[18] SPEC2000 and SPEC2006 Benchmark Suites, http://www.spec.org.