Hardware/Software Approaches for Reducing the Process Variation Impact on Instruction Fetches

ISMAIL KADAYIF, MAHIR TURKCAN, and SEHER KIZILTEPE, Canakkale Onsekiz Mart University, Turkey
OZCAN OZTURK, Bilkent University, Turkey

As technology moves towards finer process geometries, it is becoming extremely difficult to control critical physical parameters such as channel length, gate oxide thickness, and dopant ion concentration. Variations in these parameters lead to dramatic variations in access latencies in Static Random Access Memory (SRAM) devices. This means that 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 during instruction fetches when both address translation (instruction Translation Lookaside Buffer (TLB) access) and instruction cache access take place, making this solution infeasible for future high-performance processors. In this study, we first propose some hardware and software enhancements and then, based on those, investigate several techniques to mitigate the effect of process variation on the instruction fetch pipeline stage in modern processors. For address translation, we study an approach that performs the virtual-to-physical page translation once, then stores it in a special register, reusing it as long as the execution remains on the same instruction page. To handle varying access latencies across different instruction cache lines, we annotate the cache access latency of instructions within themselves to give the circuitry a hint about how long to wait for the next instruction to become available.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Process variation, instruction cache, address translation, encoding

ACM Reference Format:
Kadayif, I., Turkcan, M., Kiziltepe, S., and Ozturk, O. 2013. Hardware/software approaches for reducing the process variation impact on instruction fetches. ACM Trans. Des. Autom. Electron. Syst. 18, 4, Article 54 (October 2013), 23 pages.
DOI: http://dx.doi.org/10.1145/2489778

1. INTRODUCTION

Over the last three decades, scaling of Complementary Metal Oxide Semiconductor (CMOS) devices has improved the performance of computer systems dramatically. However, concurrent with finer-granular process technologies, it has becoming increasingly difficult to keep transistor quality within desired bounds. This problem is especially evident in sub-50nm and deeper regimes; as a result, process variation [Alam 2008] is emerging as an important issue for future designs. Process variation can be defined as the deviation from intended or designed target values of a circuit parameter of

Authors' addresses: I. Kadayif, M. Turkcan, and S. Kiziltepe, Department of Computer Engineering, Canakkale Onsekiz Mart University, Canakkale 17100, Turkey; email: kadayif@comu.edu.tr; O. Ozturk, Department of Computer Engineering, Bilkent University, Bilkent, Ankara 06800, Turkey; email: ozturk@cs.bilkent.edu.tr.

DOI: http://dx.doi.org/10.1145/2489778

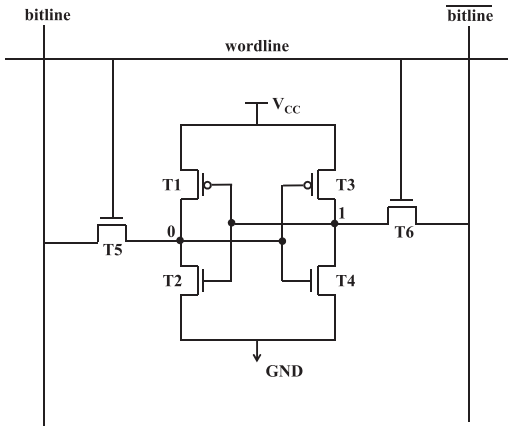

Fig. 1. Six-transistor SRAM cell storing one bit.

concern, such as channel length or width, gate oxide thickness, and random placement of dopants in a channel. It leads to significant variability in chip performance, power consumption and stability [Borkar et al. 2003; Chang and Sapatnekar 2005; Fu et al. 2009]. Such variations may happen across identically designed neighboring devices (intradie variation) [Agarwal et al. 2002] as well as across different identically designed chips (interdie variation) [Nassif 2001].

As technology scales, intradie variations are particularly important because they are increasing, and their primary source is the random placement of dopants [Tang et al. 1997]. Caused by difficulties in deep submicron process technologies, the random placement of dopants is independent of transistor spatial locality and can lead to a threshold voltage mismatch among transistors in the same hardware component. Since SRAM structures constitute a significant portion of a die area (for instance, for Alpha 21264 and Strong ARM, 30% and 60% of the die areas, respectively, are devoted to cache structures [Manne et al. 1998]) and are typically designed with minimum-sized transistors for density reasons [Papanikolaou et al. 2005], they are quite prone to random placements of dopants. As a result, different devices in the same cache line may have different performance and reliability characteristics. Frequently accessing system components in the instruction fetch stage and delayed access to the instruction cache and the instruction TLB due to process variation can degrade the system performance significantly. We believe that, in addition to circuit solutions [Chen and Naffziger 2003; Gregg and Chen 2007; Tschanz et al. 2002], architecture and compiler-based solutions to tackling the performance ramifications of process variation in the instruction fetch stage are important [Papanikolaou et al. 2005].

Although threshold voltage mismatch can also cause read/write stability failures [Agarwal et al. 2005b; Chen et al. 2005], we confine our study to access time failures (performance effects) of threshold voltage fluctuations of neighborhood transistors in SRAMs. A typical six-transistor cell used for SRAMs is depicted in Figure 1. 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, $\Delta_{MIN} \approx 0.1V_{CC}$) 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 (V_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, cache/TLB access latency will not be uniform; that is, the different cache lines/TLB entries may have different access latencies. A straightforward solution to this problem for the instruction cache is to adopt the worst-case access latency paradigm in the design; the same argument also goes for the TLB. This means all the cache lines are assumed to have the latency of the slowest cache line. While this assumption makes the design simple, it may introduce a significant performance penalty, considering the fact that both the cache and the TLB are accessed very frequently. This performance penalty is expected to increase in the move to the finer process technologies [Bowman et al. 2002; Zuchowski et al. 2005].

In this study, we investigate various hardware and software solutions to the problem of performance overheads on instruction fetches stemming from process variation in modern processors. To handle delay violations during address translations, we place a special register between the TLB and the CPU. This register, called the *Current Frame Register* (CFR), holds the last virtual-to-physical address translation and prevents access to the TLB if the translation demanded by the next instruction access is the one captured by the register. Since instruction accesses are known to exhibit high levels of locality, one can expect this register to supply the required address translations most of the time. Consequently, even if we assume the worst-case access latency for those accesses that miss the CFR and go to the TLB, the overall impact on performance will not be very high because the number of those accesses would be very low.

On the other hand, to tackle the latency discrepancies over the different cache lines, we make use of a compiler analysis to build the control flow graph (CFG) of the application and annotate the cache access latency information of each succeeding instruction. For this purpose, some unused bit positions of the instructions can be used to encode the latencies. As soon as an instruction is fetched from the cache at the pipeline's instruction fetch stage, these bits are decoded to obtain the cache access latency information, which defines how long to wait to access the cache for the succeeding instruction. Note that an alternate option could be to maintain a table storing the latencies. We could index the latency table using the predicted set address to obtain the cache access latency so that the dependent instructions could be fetched accordingly. For two main reasons we did not choose this alternate option: first, the table would introduce some area and power ramifications; second, the table itself may be subjected to process-variation-related effects.

We implemented a number of techniques based on the two basic concepts just mentioned within a simulation platform and performed experiments on codes in SPEC2000 and SPEC2006 suites. Our evaluation indicates that, when we work with a cache experiencing access variations due to process parameter variations, the proposed techniques achieve significant performance benefits over both the worst case latency assumption and an alternate scheme that uses an oracle to determine the latency of a TLB entry/cache line to be accessed ahead of time. For example, for the case where 15% of cache lines and TLB entries are assumed to be affected by process variation, the average performance loss under the worst-case latency paradigm assumption is around 25.6%. The performance loss for the same system is 6.1% if the hardware has prior knowledge of whether the cache and TLB accesses would be in perfect/imperfect (process variation affected/unaffected) entries and is thus able to adjust the timing of pipeline stages accordingly. On the other hand, our hardware-managed CFR (HMCFR) for address-translation- and code-relocation-based encoding (CRBE) for cache access techniques can together reduce the performance loss in the instruction fetch stage to 2.1%.

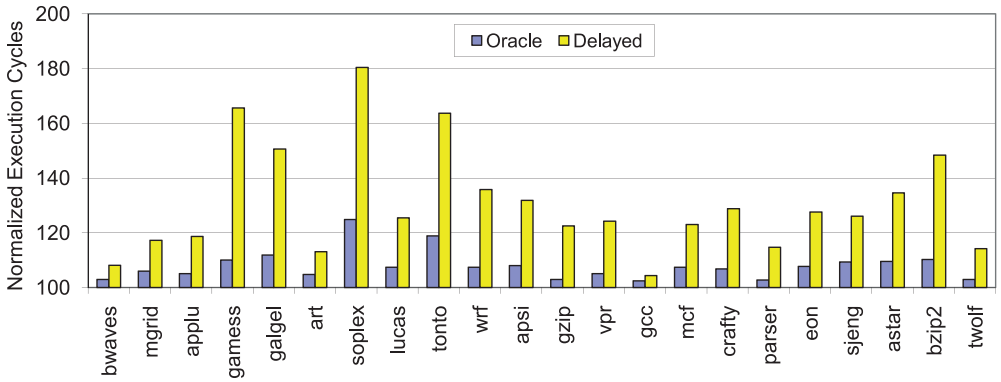

Fig. 2. Normalized execution cycles for oracle and worst-case assumption cases. All values are normalized with respect to the corresponding values for the ideal case, where it is assumed that the cache is not affected by process variation. The results are for the case where 25% of the cache lines are affected by process variation and while the access to a perfect cache line completes in a single cycle, the access to an imperfect cache line completes in 2 cycles.

The rest of this article is structured as follows: in the next section, we try to explain why the worst-case access latency paradigm cannot be accepted under process variation. In Section 3 we introduce our basic hardware and software enhancements, which are necessary to implement our schemes. We introduce our schemes in Section 4. The experimental setup is given in Section 5. We evaluate our schemes in Section 6. Related work is summarized in Section 7. Finally, our concluding remarks are given in Section 8.

2. EGALITARIAN CACHE MANAGEMENT UNDER PROCESS VARIATION

To quantify the performance overhead in execution time introduced by process variation, let us look at Figure 2. This figure indicates the performance degradation when the cache is assumed to be affected by process variation. The results in the graph are with respect to the performance values of the ideal case, where it is assumed that the cache is immune to process variation. The first bar for each application in the figure (Oracle) plots the performance value of a scheme with an oracle predictor, which can foretell for each cache access whether the corresponding cache set is affected by process variation. In these experiments, we assume that when we access a line in an imperfect set, the access takes an extra cycle (completing in two cycles); otherwise, the access completes in one cycle. By an "imperfect set" of the cache we mean the set including at least one cache line with an access failure (delay violation) due to parameter variation. An "imperfect entry" of the TLB stands for an entry whose access takes longer because of process variation. The results are given for a cache where 25% of the lines are imperfect. Our performance results are based on the assumption that a specific percentage of cache lines are affected by process variation and such lines are randomly distributed over the cache as in Mutyam and Narayanan [2007]. The second bar plots the performance value of the access mechanism based on the worst-case access latency paradigm (Delayed); where each cache access takes two cycles. We see from this figure that the average performance losses from the Oracle and Delayed schemes are approximately 8.1% and 30.2%, respectively. We also observe that in some cases, such as with gamess, soplex, and tonto, the overhead resulting from the scheme based on the worst-case assumption results in more performance degradation of more than 60% .

To see the performance effects of process variations on the cache with a larger access delay, we have repeated the experiments whose results are given in Figure 3. In these

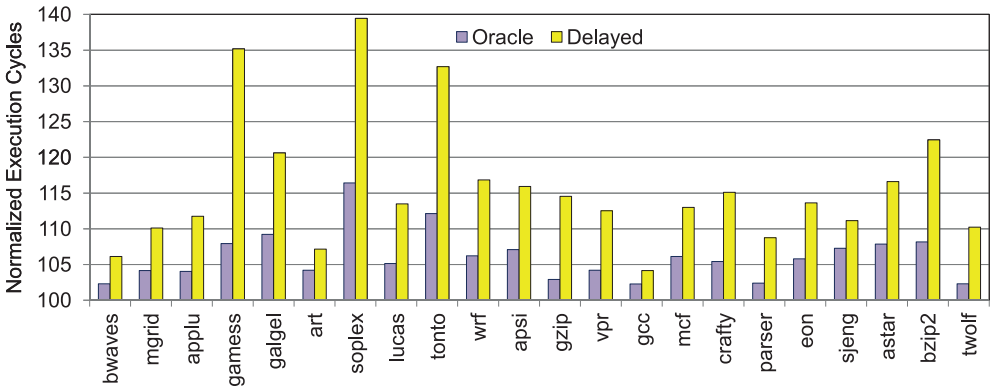

Fig. 3. Normalized execution cycles for a cache with multi cycle access latency. The results are for the case where 25% of the cache lines are affected by process variation and while the access to a perfect cache line completes in 3 cycles, the access to an imperfect cache line completes in either 4 or 5 cycles, depending on the severity of variations.

experiments, we assume that while a perfect cache line access completes in 3 cycles, an imperfect cache line access may take either 4 or 5 cycles, with an equal probability, depending on the severity of variations. The average performance degradation from the Oracle and Delayed schemes are around 6.0% and 15.9%, respectively. When Figures 2 and 3 are considered together, it can be concluded that the performance losses due to process variation are less severe for caches with a larger access latency. Moreover, the results of the Delayed scheme motivated us to investigate new techniques for reducing performance loss due to process variation. The results of the Oracle scheme show that our proposed techniques of alleviating performance loss introduced by process variation are quite effective.

3. OUR PROPOSED HARDWARE AND SOFTWARE ENHANCEMENTS

In this section we introduce our basic hardware and software enhancements proposed for tackling delay discrepancies due to process variation in the instruction fetch stage. We will first look at TLB-oriented modifications and then cache-oriented modifications.

3.1. Generating Physical Addresses Directly

Fast address translation has been the subject of previous studies [Knight and Rosenfeld 1984; Maddock et al. 1981; Chiueh and Katz 1992; Strecker 1978]. To facilitate address translation in modern microprocessors, a special structure called a Translation Lookaside Buffer (TLB) is used. TLB is a cache that stores recent virtual-to-physical translations of instruction pages.

On the other hand, a cache lookup requires indexing a set and a subsequent tag comparison across the lines within that set. Indexing and tag comparison can be performed using either a virtual address or a physical address, leading to four possible configurations. Since virtually indexed, physically tagged (VI-PT) caches are more common, in this study we focus only on those. However, it is possible to extend our work to handle other cache lookup schemes as well.

In a VI-PT cache lookup, the virtual address is used to index the cache, and the TLB is concurrently looked up to obtain the physical address, which removes the TLB from the critical path. Consequently, the tag from the physical address is used for comparison with the tags of the blocks within the chosen set. In modern processors, a cache access and a TLB access is expected to complete within the same cycle for a nonpipelined cache, which is very important from the system's performance viewpoint.

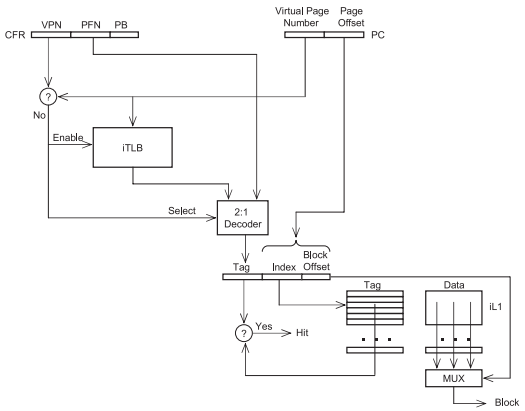

Fig. 4. L1 instruction cache lookup using the CFR under the VI-PT cache lookup mechanism.

However, imperfect TLB entries (entries that are affected by process variations during the manufacturing process) may cause the TLB access to complete later than the cache indexing, thereby delaying the tag comparison and in turn the data access in the selected set of the cache.

To avoid regenerating the last page translation we use a Current Frame Register (CFR) in our design. The CFR can be considered a special register and is assumed to be unaffected by process parameter variation. This can be ensured by including a few additional registers in the design and by selecting a fault-resilient one as the CFR. This register keeps the current virtual-to-physical translation. As long as the control flow remains within the current instruction page, the translation can be safely obtained from the CFR rather than the TLB. Thus, we do not pay any performance penalty for accessing the TLB with nonuniform access latencies. Our use of the CFR is similar to the one introduced in Kadayif et al. [2002, 2007], where it was used to optimize the TLB's dynamic energy. In this article, we exploit the CFR register in the context of mitigating the impact of process variations on TLB latency. The basic format of the CFR is as follows:

[<Virtual Page Number> <Physical Frame Number> <Protection Bits>].

We assume that the physical frame number and the protection bits of the corresponding TLB entry are kept in the CFR. The next instruction fetch with the CFR under the VI-PT L1 lookup mechanism is shown in Figure 4. To start page translation early when the translation does not exist in the CFR, for each translation the TLBs are looked up with the virtual page number (VPN) of the PC (program counter). At the same time, the VPNs of the CFR are compared to those of the PC. If they match, the TLB lookup is canceled and the physical frame number (PFN) in the CFR creates the tag portion of the physical address, which is used to compare the tags in the set that was indexed in the cache. Otherwise, the translation coming from the TLB lookup creates the tag portion of the physical address. Since the program execution flow leaves the current page boundary very infrequently, one can expect that most of the time the translation will be found in the CFR register, hence overcoming access time violations caused by imperfect TLB entries.

3.2. Minimizing the Number of Imperfect Cache Sets and Encoding Cache Access Latencies

Our cache-oriented schemes rely on compiler to encode the latency information for cache lines. Since at compilation time, in general, only the set (not the cache line)

in which an instruction will 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). It is clear that the success of our cache-oriented schemes largely depends on the number of cache sets affected by process variations. Thus, it is very important to minimize the number of imperfect sets.

To minimize the number of imperfect cache sets, we introduce a technique called *line reshuffling*, which is similar to *block rearrangement techniques* proposed by Mutyam and Narayanan [2007]. They considered block rearrangement either between a pair of two adjacent cache sets or among all cache sets. On the other hand, we perform line reshuffling among a specific number of cache sets, which can be implemented by a programmable address decoder [Shirvani and McCluskey 1999]. In this technique, the control inputs of the pass transistors driving the word lines are programmed in such a way that any set is allowed to include either only perfect cache lines or only imperfect cache lines 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,

$$2^r j \leq i < 2^r(j+1), \textit{where } 0 \leq j < s/2^r.$$

For reshuffling we need to know which lines are perfect and which ones are imperfect. To do so, we can employ the March test [Chen et al. 2005], 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 perfect, $f_i = 0$; or as imperfect, $f_i = 1$.

While Figure 5(a) indicates a reshuffling with the reshuffling degree of 2, Figure 5(b) shows the last two level decode logic in the address decoder necessary for reshuffling. Here, the reshuffling is applied to four consecutive cache lines by considering the two least significant set bits in address bits as well as the access latencies of the cache lines in question. In reshuffling process, lines in a reshuffling group rearranged in such a way that the perfect cache lines are moved toward the top of the group while the imperfect cache lines are moved toward the bottom of the group. For example, assume that the access latencies of four lines (l_0, l_1, l_2, l_3) in a reshuffling group correspond to $(f_0, f_1, f_2, f_3) = (1, 0, 1, 0)$. After reshuffling, l_3, l_0, and l_1 map to l_0, l_1, and l_3, respectively, as shown in Figure 5(a), which can be easily validated by examining logic equations in Figure 5(b). This means that, for example, the addresses originally mapping into line l_3 will map into l_0. In this way, the addresses whose the least two significant bits are 00 and 01 are mapped into perfect lines l_1 and l_3 in the corresponding reshuffling group while the addresses whose the least two significant bits are 10 and 11 are mapped into imperfect lines l_2 and l_0. To limit the area overhead due to reshuffling, we have performed reshuffling among 8 lines in our study. Although we have not calculated the area overhead, with a limited reshuffling degree such as 3 it is expected to be small as demonstrated in Shirvani and McCluskey [1999].

In our hardware, we assume that there is a buffer between the cache and the CPU that has the capacity of storing one cache block. As long as the requested instructions are available in this buffer, the demands are satisfied from there. Otherwise, a cache access is made, and the whole cache block storing the instruction in question is read out of the cache and is stored in this buffer for the next instruction fetch. We further assume that there are some unused (empty) bit slots or reserved

f_0	f_1	f_2	f_3	l_0	l_1	l_2	l_3
0	0	0	0	l_0	l_1	l_2	l_3
0	0	0	1	l_0	l_1	l_2	l_3
0	0	1	0	l_0	l_1	l_3	l_2
0	0	1	1	l_0	l_1	l_3	l_2
0	1	0	0	l_0	l_3	l_1	l_2
0	1	0	1	l_0	l_3	l_1	l_2
0	1	1	0	l_0	l_3	l_2	l_1
0	1	1	1	l_0	l_3	l_2	l_1
1	0	0	0	l_3	l_0	l_1	l_2
1	0	0	1	l_3	l_0	l_1	l_2
1	0	1	0	l_3	l_0	l_2	l_1
1	0	1	1	l_3	l_0	l_2	l_1
1	1	0	0	l_3	l_2	l_0	l_1
1	1	0	1	l_3	l_2	l_0	l_1
1	1	1	0	l_3	l_2	l_1	l_0
1	1	1	1	l_3	l_2	l_1	l_0

(a)

line	Last two level decode logic
l_0	$\overline{f_0}\ \overline{a_0}\ \overline{a_1} + f_0 a_0 a_1$
l_1	$\overline{f_0}\ \overline{f_1} a_0 \overline{a_1} + \overline{f_0} f_1 a_0 a_1 + f_0 \overline{f_1}\ \overline{a_0}\ \overline{a_1} + f_0 f_1 \overline{a_0} a_1$
l_2	$\overline{f_0}\ \overline{f_1}\ \overline{f_2}\ \overline{a_0} a_1 + \overline{f_0}\ \overline{f_1} f_2 a_0 a_1 + \overline{f_0} f_1 \overline{f_2} a_0 \overline{a_1} +$ $\overline{f_0} f_1 f_2 \overline{a_0} a_1 + f_0 \overline{f_1}\ \overline{f_2}\ a_0 \overline{a_1} + f_0 \overline{f_1} f_2 \overline{a_0} a_1 +$ $f_0 f_1 \overline{f_2}\ \overline{a_0}\ \overline{a_1} + f_0 f_1 f_2 a_0 \overline{a_1}$
l_3	$\overline{f_0}\ \overline{f_1}\ \overline{f_2} a_0 a_1 + \overline{f_0}\ \overline{f_1} f_2 \overline{a_0} a_1 + \overline{f_0} f_1 \overline{f_2}\ \overline{a_0} a_1 +$ $\overline{f_0} f_1 f_2 a_0 \overline{a_1} + f_0 \overline{f_1}\ \overline{f_2}\ \overline{a_0} a_1 + f_0 \overline{f_1} f_2 a_0 \overline{a_1} +$ $f_0 f_1 \overline{f_2} a_0 \overline{a_1} + f_0 f_1 f_2 \overline{a_0}\ \overline{a_1}$

(b)

Fig. 5. Line reshuffling and decode logic for reshuffling degree r = 2. (a) Reshuffling among four lines based on their access latencies. Cache line i is characterized either as perfect, $f_i = 0$; or as imperfect, $f_i = 1$. (b) Last two level decode logic in the address decoder of cache necessary for implementing reshuffling. a_0 and a_1 indicate the two least significant set index bits.

opcodes in the instruction set architecture (ISA), which can be used for encoding the latencies. Most CPUs are now 64-bit architectures, which provide a very large number of bits to encode instructions as well as a few unused bit slots or some reserved opcodes (for future usage). We can exploit these slots to encode access latencies. To encode access latencies, the compiler first builds a control flow graph (CFG) of the application and then maps instructions into the sets of the cache, based on the instruction virtual addresses. This can be done under the assumption that the page-offset bits of the virtual address constitute the cache-index bits, which is valid for most processors with the VI-VP cache access mechanism. For n different access latencies, we need to use $logn$ unused bit slots or n reserved opcodes in the instruction format.

4. SCHEMES UNDER CONSIDERATION

In this section, we first introduce some base schemes that do not exploit our proposed hardware and software enhancements. These schemes will serve for comparison. Later, we will introduce various schemes that take advantage of our hardware and software enhancements and quantitatively compare their performances with those of the base schemes.

4.1. Base Schemes

—*Perfect.* This scheme represents the ideal case, where TLB and cache structures are immune to process variation. That is, it is assumed that neither the TLB nor the cache has an imperfect entry. Each TLB entry is assumed to take the same amount of time to be accessed. The same is assumed for the cache. Thus, the TLB access and cache access complete in the same cycle.

—*Delayed.* This captures the case where the TLB and cache experience process variations, that is, the TLB/cache has some imperfect entries, which require more time to be accessed compared to the perfect entries. To resolve access latency discrepancies across the different entries, this scheme assumes the worst-case access latency in the design; that is, it assumes the access time of the slowest TLB entry/cache line for all TLB entries/cache lines. Here, we consider that the TLB access and the cache access complete within two cycles during the instruction fetch stage. The experimental results of this scheme are important because they can help justify the necessity of our hardware and software mechanisms to diminish the effect of process variations on performance.

—*Oracle.* This scheme is hypothetical rather than practical, with an oracle predictor that can foretell whether each virtual-to-physical translation resides in an perfect entry or not. We also have prior knowledge as to whether the cache block to be accessed resides in a perfect set or not. According the characteristics of the TLB and cache entries accessed, the hardware is assumed to be able to adjust the timing of the pipeline stages. If the translation is in a perfect TLB entry and the cache line belongs to a perfect set, the TLB and the cache accesses are performed in the same cycle; otherwise, they take one more cycle (for a total of two cycles) to complete. In other words, delay violations either in the TLB or in the cache slow the instruction fetch stage by one cycle. The results for this scheme can help us justify the effectiveness of our proposed schemes on alleviating the performance degradation introduced by process variation.

4.2. TLB-Oriented Schemes

4.2.1. Hardware-Managed CFR (HMCFR). In this scheme, the CFR register is managed by the hardware. One CFR register is available in the hardware, and it maintains the last instruction page's translation. As long as the execution flow of the program remains within the same page, the translation is obtained from the CFR. Otherwise (i.e., if the execution leaves the current page), we incur a TLB access, and due to the disparities between their latencies (as a result of process variation), the address translation and instruction cache access are assumed not to complete in the same cycle (as in the Delayed scheme). When the execution leaves the current page boundary, we also need to update the CFR with a translation obtained from the TLB. Note that the performance overhead of this update can be hidden from the critical path by updating the CFR as soon as the PC is updated (before the subsequent instruction fetch cycle). To start the page translation early (in cases when the translation does not exist in the CFR), for each translation a TLB access is started up concurrently with the CFR access. If the translation is found in the CFR, the TLB access is canceled immediately.

4.2.2. Software-Managed CFR (SMCFR). As mentioned before, when the execution leaves the current page boundary, an TLB lookup is triggered for the target page. This triggering is initiated when the virtual page number stored in the CFR and the one captured by the PC are not the same. At the same time, we record the translation obtained from the TLB lookup in the CFR for future use. In the HMCFR scheme, because of the performance concern, for each address translation a TLB lookup is started because we do not know in advance whether the translation exists in the CFR. On the other hand,

the SMCFR scheme determines where the address translation is to be obtained from (either the CFR or the TLB). To accomplish this, the compiler first extracts the control flow information from the code and builds the corresponding control flow graph. Later, the compiler examines the CFG and when it is certain that the next instruction to be fetched is within the current instruction page boundary, it assures the hardware that the address translation exists in the CFR. Otherwise, the compiler can insert a special instruction in the code to trigger a TLB lookup for the required page translation. The disadvantage of this approach is an increase in code size due to the special instruction insertions. To overcome these extra instruction insertions, the compiler can tell the hardware whether or not a TLB lookup is needed by exploiting an unused bit slot in the ISA. We can also use such an unused bit slot to tell the hardware whether a TLB access is needed for the address translation of the next instruction. For example, encoding a 0 (1) can signal the hardware to obtain the address translation for the succeeding instruction from the CFR (or from the TLB). It needs to be noted that there are two ways a program execution can move from one instruction page to another: first, through explicit branch instructions, whose target may be on a different page, and second, because of two successive instructions that happen to fall on page boundaries; that is, one is the last instruction on a page, and the next is the first instruction on the next page. In both cases, the compiler can encode a 1 into the preceding instruction's corresponding bit position to trigger the required translation from the TLB. Otherwise, a 0 is encoded to tell the hardware to obtain the translation from the CFR. To obtain the address translation from the CFR whenever possible, the compiler should analyze the static branches, whose target addresses can be decided at compilation time, and if the target address stays within the current page boundary, the compiler should make sure that the translation goes through the CFR. SMCFR needs a hardware interlock to always do address translation through TLB on first instruction of interrupt or exception handler and first instruction after mispredicted branch.

This compiler technique has two advantageous properties. First, unlike the CFR scheme, a TLB access is not necessary when it is determined that a translation exists in the CFR. This aspect reduces the number of TLB accesses, which in turn leads to dynamic energy savings. Second, as long as the translation is obtained from the CFR, this compiler scheme can be tailored to save TLB leakage energy by changing the power state of the TLB to a state preserving to a low-power mode.

4.3. Cache-Oriented Schemes

Our cache-oriented techniques rely on compiler to encode the latency information for cache lines. As suggested in Mutyam and Narayanan [2007], a latency table can be used to store the latency information of cache sets. In our case a single bit for each cache set is used to mark it either perfect or imperfect. This table can be initialized through the March test before the operational phase of a microprocessor whose instruction cache is subjected to process parameter variations. The compiler can use this table to obtain the latency information of cache sets prior to compilation.

We explain each of our cache-oriented schemes with the sample CFG given in Figure 6. For simplicity, in the figure, we assume that each cache block can store four instructions. For example, instructions $I_1, I_2, I_3,$ and I_4 in basic block B_1 constitute a cache block, which maps into a cache line in set s_1. Up to now, we have implicitly assumed that each instruction encodes the succeeding instruction's access latency. However, since in our design we assume that the whole cache block is read out of the cache and stored in the buffer (see earlier), we only need to encode access latencies in two different types of instructions: first, in the last instruction of a cache block to encode the succeeding set's access latency (for instance, instructions I_4 and I_{12}); and second, in the instructions modifying the program's execution flow, whose target

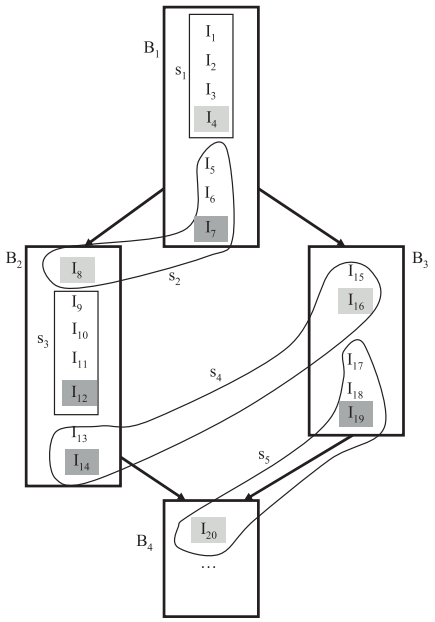

Fig. 6. Sample CFG.

instruction is beyond the cache block boundary (for instance, instructions I_7 and I_{14}). As mentioned earlier, it is possible to encode n different access latencies by using $logn$ empty bit slots; for simplicity, in our schemes we encode two different access latencies using one bit. In the figure, we assume that the light-shaded instructions encode the access latency of the cache lines that map into a perfect cache set, whereas the dark-shaded instructions encode the access latency of the cache lines that map into an imperfect cache set. Our cache oriented schemes need a hardware interlock to always allow two cycles for the first instruction cache access of interrupt or exception handler.

Note that we can also apply our mechanisms to library codes. If libraries are statically linked, the compiler can go through the library code to annotate the cache access latency of the next instruction as in an application code. If libraries are dynamically linked, in this case dynamic recompilation can be applied.

4.3.1. Conservative Encoding (CE). In this scheme, the compiler does not carry out any analysis, and the target of a branch instruction is conservatively assumed to fall into an imperfect cache set; this assumption is made in Figure 6. Based on this scheme, branch instructions I_7, I_{14}, and I_{19} are encoded with the access latency of the imperfect sets, as shown in the figure (that is, they are dark shaded).

4.3.2. Less-Conservative Encoding (LCE). In this scheme, the compiler analyzes the target of each branch instruction. If the fall-through and the target instructions fall into perfect sets, the branch instruction in question is encoded with the access latency of the perfect sets. Otherwise, the access latency of the imperfect set is encoded into the branch. Although the LCE scheme is better than the CE scheme, it is still conservative because the compiler may not locate the target instruction of some branches in the binaries (for instance, a branch instruction whose target is taken from a register). Fortunately, the number of these branch instructions (which we call statically unanalyzable branch instructions) is not high, compared to the statically analyzable instructions. If both s_2 and s_4 are perfect sets in Figure 6, the LCE scheme will encode

Table I. Configuration Parameters and Their Values Used in Our Experiments

Processor Core	
Functional Units	8 Integer ALUs, 4 Integer mult./divide, 8 FP add, 4 FP multiply, 4 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, 1-cycle latency
L1 data cache	64KB, 4-way (LRU), 32 byte blocks, 1-cycle latency
L2 cache	8MB unified, 8-way (LRU), 128 byte blocks, 12-cycle latency
Data/Instruction TLB	128 entries, full-associative, 30-cycle miss penalty
Memory	160-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

instruction I_7 with the access latency of perfect sets. However, if either of these two sets is imperfect, instruction I_7 is encoded with the access latency of imperfect sets.

4.3.3. Code-Relocation-Based Encoding (CRBE). In this scheme, in addition to trying to locate the target of statically analyzable branch instructions (as in the LCE scheme), the compiler also tries to relocate the code so that both the fall-through and the target instructions of the branch fall into the same kind of cache sets; that is, if one falls into a perfect/imperfect set, the compiler makes an attempt to map the other into a perfect/imperfect set. We also aim to map frequently executed instructions to perfect sets so that they can be fetched as quickly as possible. We have especially focused on instructions in loop bodies with high iteration counts. We first build a control flow graph (CFG) and then apply profiling to identify the edge frequencies in the CFG. Since the compiler knows the access latencies of the sets, when it is necessary it relocates the frequently executed code such that it is mapped into the nearest cache set ahead. To do so, the compiler inserts an unconditional branch instruction in the binary whose target is the address of the first instruction in the relocated code. The address holes from the inserted branch instruction to the first instruction of the relocated code are filled with nop instructions in the binary. The preference in the code relocation process is given to the most frequently executed code blocks and it continues until the code size increase in binary reaches at a predefined threshold, which is 2% in our experiments.

5. EXPERIMENTAL SETUP

We implemented all our schemes by modifying SimpleScalar 3.0 [SimpleScalar 2012], 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 the schemes discussed in the previous section into an Alpha-like platform. All compiler analyses and code relocations were done based on pre-compiled binaries. Unless otherwise stated, the simulation parameters used in our experiments are listed in Table I. We used some codes from the SPEC2000 suite (mgrid, applu, galgel, art, lucas, apsi, gzip, vpr, crafty, parser, eon, and twolf) as well as

Table II. Benchmarks Used in Our Experiments and Their Important Characteristics

Benchmark Name	Number of TLB (Cache) Accesses	Number of Execution Cycles	Number of CFR Updates
bwaves	347110044	347982163	10918 (<0.01%)
mgrid	258785124	260407090	199747 (0.08%)
applu	260821970	261959435	868449 (0.33%)
gamess	120163487	160312282	11213959 (9.33%)
galgel	186777092	191221037	22288 (0.01%)
art	556587751	630700741	2607318 (0.47%)
soplex	109969817	130813719	8949246 (8.14%)
lucas	236223857	278925939	3945015 (1.67%)
tonto	109322546	180113099	7724247 (7.07%)
wrf	122194786	195349559	6544230 (5.36%)
apsi	189621174	210554664	1989830 (1.05%)
gzip	167916846	272520611	4140030 (2.47%)
vpr	308024576	435778113	14786342 (4.80%)
gcc	204429261	205505024	278397 (0.14%)
mcf	616617384	643506260	21447366 (3.48%)
crafty	153696081	256467119	12093625 (7.87%)
parser	237277415	371959380	18294287 (7.71%)
eon	140157315	257521967	11674993 (8.33%)
sjeng	186549445	367955497	18846428 (10.10%)
astar	143319970	212187011	12359927 (8.62%)
bzip2	168465412	231748913	5821013 (3.46%)
twolf	404191737	569302265	11311309 (2.80%)

some codes from the SPEC2006 suite (bwaves, gamess, soplex, tonto, wrf, gcc, mcf, sjeng, astar, and bzip2) [SPEC 2012] in our experiments. Since the simulation takes a long time to run the benchmarks we considered to completion, we used SimPoint [Hamerly et al. 2005; Perelman et al. 2003] to generate simulation points. For each benchmark, we fast forwarded a specific number of instructions, as suggested by Sherwood et al. [Sherwood et al. 2001], and then simulated the next 500 million instructions on predetermined simulation points. Table II gives the dynamic number of TLB/cache accesses and the execution cycles of these benchmarks when the Perfect scheme is used, under the configuration parameters listed in Table I. The values under the last column show how many times the execution control leaves the current page boundary (that is, the number of CFR updates).

6. EXPERIMENTAL RESULTS

In our experiments we assume that the percentage of TLB entries/cache lines affected by process variations varies from 15% to 25% to 40%. This range is reasonable since the circuit-level simulations show that with 32nm technology, bit-level flip rates are around 0.4% [Liang et al. 2007], which leads to a 64% probability of cache-line failure for 32-byte lines. 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.

6.1. Evaluation of TLB-Oriented Schemes

In this section we assume that only the TLB is affected by process variation; that is, the cache is immune to process variation. The execution cycle results for such a

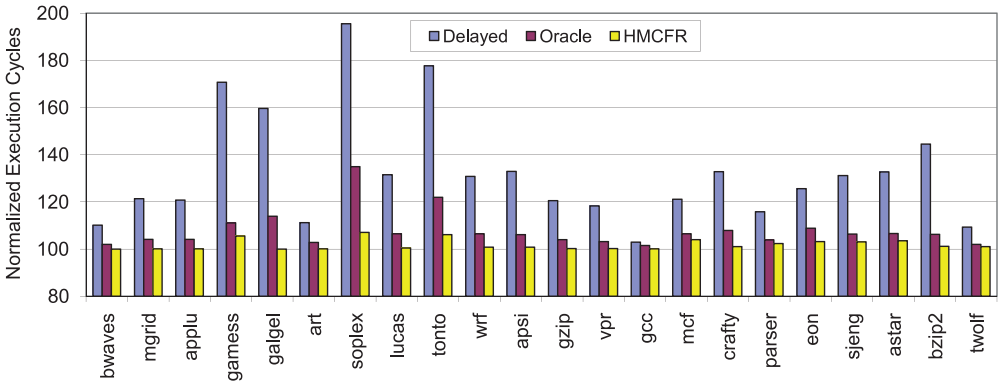

Fig. 7. Normalized execution cycles for the Delayed, Oracle, and HMCFR schemes. Only TLB is assumed to be affected by process variation, and the ratio of the affected TLB entries is 25%.

system with the Delayed, Oracle, and HMCFR schemes are plotted in Figure 7. These performance values are normalized with respect to the cycle values of the Perfect scheme, which are presented in Table II. In these experiments, 25% of the TLB entries are considered to carry process variation effects.

As can be clearly seen from the figure, delaying the TLB access conservatively (based on the worst-case access latency assumption) affects the program performance significantly, especially in benchmarks such as gamess, soplex, and tonto. For instance, the TLB access delay due to process variation increases the execution cycles for soplex and tonto as much as 95.1% and 77.2%, respectively. The significant performance degradation experienced by these three benchmarks can be attributed to their high IPC values, which causes the TLB accesses to fall into the critical path most of the time, thereby extending execution time significantly. We observe that, on average, the Delayed scheme results in 33.2% performance degradation over the Perfect scheme (the ideal scheme). The average performance degradation with the Oracle scheme is 7.7%. Note that in these experiments we assume the TLB entries affected by process variation are randomly distributed over the entire TLB space. These results tell us that even if we are able to perfectly predict the latency of the TLB entry to be accessed next (and adjust the execution to take advantage of this knowledge), the incurred performance penalty due to process variation will still be high.

When we look at the performance values for HMCFR on the graph, we can easily see that this scheme performs much better than the Delayed and Oracle schemes for all the benchmarks. The worst performance experienced by the HMCFR scheme is for the benchmark soplex, and it is around 7% worse than that of the Perfect scheme, which represents the ideal case. More importantly, for most of the benchmarks, the performance of HMCFR is within 1% of the Perfect scheme. The reason why our scheme performs better than the other schemes (except for the ideal scheme with no process variation) is the fact that most of the time we are able to find the requested virtual-to-physical page translation in the CFR register (See the last column of Table II), which lists the number of CFR updates required during program execution. Finding the translation in the CFR removes the need for TLB access and allows the cache and TLB accesses to complete in the same cycle. Note that by employing more CFR registers (instead of our default 1), it may be possible to further improve performance by catching more translations. Further investigation of this issue is beyond the scope of this article.

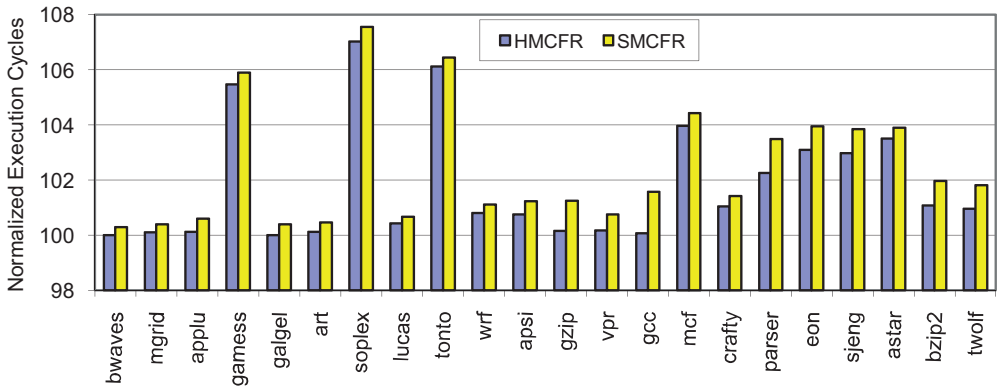

Fig. 8. Normalized execution cycles for the HMCFR scheme and the alternate SMCFR scheme. All values are normalized with respect to those of the Perfect scheme. Only TLB is assumed to be affected by process variation, and the ratio of the affected TLB entries is 25%.

6.1.1. Comparison of HMCFR and SMCFR. The comparison of the HMCFR and SMCFR schemes is depicted in Figure 8 in the context of execution cycles. Here the results reflect the assumption that only the TLB is affected by the process variation (the cache is immune to process variation). As depicted in the figure, the HMCFR scheme performs better than the compiler-based scheme for each benchmark tested. The compiler-based scheme performs worse because the compiler cannot determine (statically) whether the target of some branches should stay on the current page. Therefore, it obtains the address translations for those branch target instructions from the TLB instead of the CFR, which leads to some performance degradation. This degradation is more pronounced in integer-based applications, as evident from the graph. The average performances of the HMCFR scheme and the SMCFR scheme are 1.8% and 2.4% lower than that of the Perfect scheme with a page size of 8K. Since HMCFR performs better than SMCFR, for the rest of our results we will not consider SMCFR.

6.1.2. Sensitivity to Page Size. To assess the robustness of the HMCFR scheme, we measured its sensitivity to different instruction page sizes. For this purpose, we also conducted experiments with page sizes of 4K and 16K. These results are presented in Figure 9. In this graph, for each benchmark, the first, second, and third bars correspond to normalized execution cycles with respect to the Perfect scheme when we use instruction page sizes of 4K, 8K, and 16K, respectively. As seen in the figure, it is possible to improve performance by increasing page size, especially of those benchmarks incurring large performance overheads through the HMCFR scheme. For example, for soplex, we can cut the performance overhead from 8.6% to 7.1% to 5.3% by increasing page size from 4K to 8K to 16K. The average overhead incurred by the HMCFR scheme with a page size of 4K is 2.2%, whereas the average overheads with page sizes of 8K and 16K are 1.8% and 1.3%, respectively. On the other hand, the harmonic means of the performance overheads are around 2.1%, 1.7% and 1.3%, respectively. Since we did not observe any significant change in our results when the instruction page size was increased beyond 16K, we do not present the results of larger page sizes.

6.2. Evaluation of Cache-Oriented Schemes

In this section, we assume that only the cache is affected by process variation; that is, the TLB is immune to process variation. In all our evaluations, reshuffling degree is assumed to be 3; that is, reshuffling is implemented among eight consecutive cache lines.

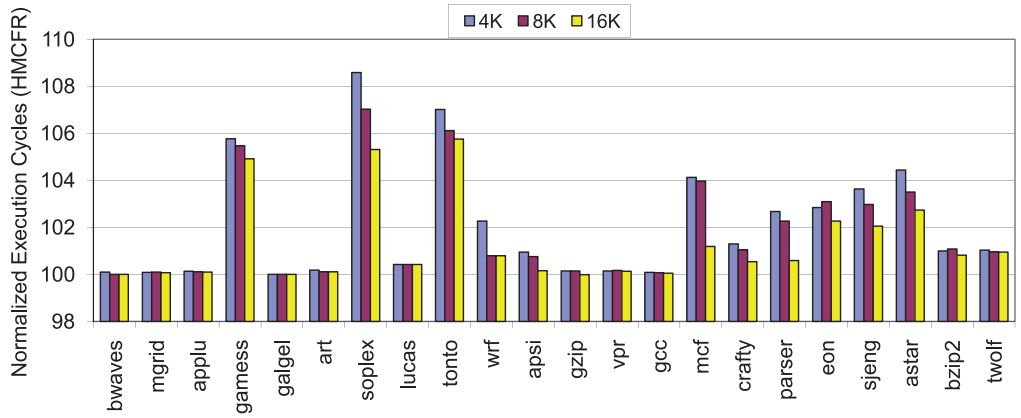

Fig. 9. Normalized execution cycles for the HMCFR scheme under page size of 4K, 8K, and 16K. All values are normalized with respect to the corresponding values for the Perfect scheme. 25% of TLB entries are assumed to be affected by process variation.

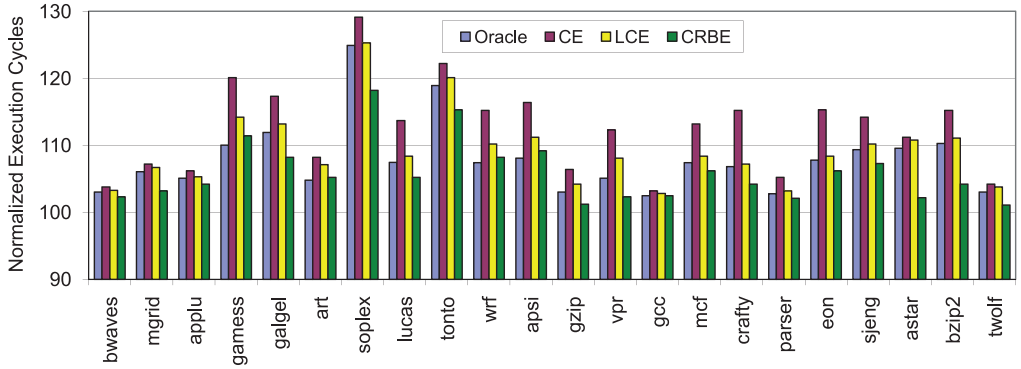

Fig. 10. Normalized execution cycles for various cache access latency encoding schemes under the assumption that 25% of the cache lines are affected by process variation. All values are normalized with respect to the corresponding values for the Perfect scheme. Only the cache is assumed to be affected by process variation (the TLB is immune to the variation).

6.2.1. Comparison of Our Cache-Oriented Schemes. The bar chart in Figure 10 presents the performance values of various techniques aiming at alleviating the effects of process variation on the cache. For any benchmark, each bar from left to right shows the performance values for the Oracle, CE, LCE, and CRBE schemes, respectively. These performance values are normalized with respect to the values of the Perfect scheme and reflect the case where 25% of the cache lines are affected by process variation. Process variation effects are considered to be randomly distributed across the cache lines and these effects extend the access latency of the sets by one cycle, causing a load to complete in two cycles. Since we consider buffered stores, process variation does not introduce any performance overhead for stores, as long as the buffer can accommodate the stores.

From this figure, we observe that the average (harmonic means) performance cycles across all benchmarks for the CE and LCE schemes are 12.5% (12.1%) and 9.2% (9.0%) above the cycles of the Perfect scheme, respectively, whereas the average (harmonic means) performance cycles for the CRBE scheme is around 5.9% (5.7%) above the cycle values of the Perfect scheme. Considering the average values from the Delayed and

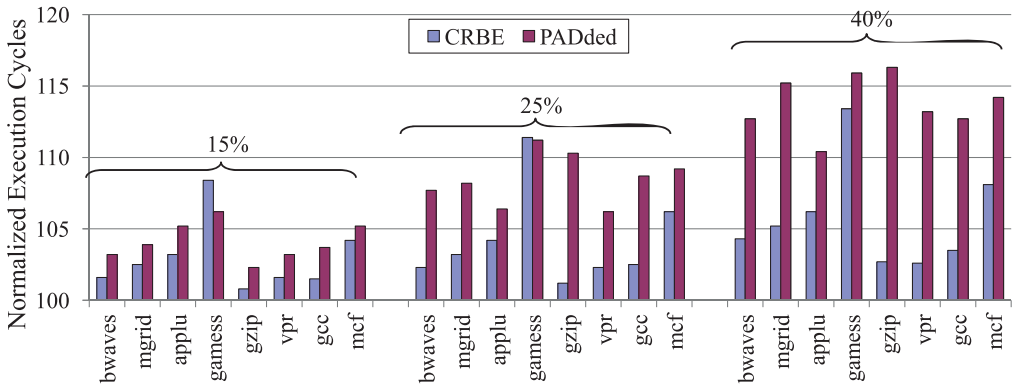

Fig. 11. Normalized execution cycles for various percentages of affected cache lines for the CRBE scheme and the PADed cache. The percentages of affected lines vary from 15% to 25% to 40%.

Oracle schemes (30.2% and 8.1%), we reach the following conclusions. First, the large overhead (30.2%) introduced by the Delayed scheme (adopting the worst-case access latency paradigm) necessitates new architectural-level mechanisms for caches affected by process variations. Second, encoding access latencies in the instructions themselves, even without analyzing targets of the branch instructions, improves performance dramatically, cutting the overhead to 12.5% on average. Third, with the LCE scheme, the average performance overhead (9.2%) comes very close to that of the Oracle scheme (8.1%), which means that encoding access latencies based on analyzing targets of the branches is very effective. Fourth, by encoding access latencies and applying code relocation, it is possible to drop the overhead (to 5.9%) to below the one introduced by the Oracle scheme, which is based on the assumption of a completely accurate prediction of set access latencies. This clearly shows the effectiveness of code relocation in alleviating the effect of process variation on performance. Also, the code size increase for the CRBS scheme due to inserting unconditional branches for code relocation is under 1%, and therefore negligible, so can be omitted.

6.2.2. Comparison of CRBE and PADded Cache. The PADded cache was presented as a fault-tolerance technique for cache memories by Shirvani and McCluskey [1999]. This technique employs a special Programmable Address Decoder (PAD) to disable faulty cache lines and to re-map their references to healthy lines; thus, tolerating faulty cache lines. It was claimed that PADed caches could be a very effective way of tolerating permanent faults in cache lines and introduce around 10% area overhead. This technique can also be employed to handle longer access latencies by diverging imperfect line references into perfect cache lines. To compare our CRBE scheme to the PADded cache technique, we conducted a set of experiments. In CRBE experiments, reshuffling was implemented among 8 lines. Similarly, in PADded cache experiments re-mapping was implemented among the cache lines of 8 consecutive cache sets in each way. Furthermore, we uniformly distributed the references of imperfect cache lines to perfect cache lines in each 8 consecutive sets. For example, if 4 out of 8 lines are imperfect, the references to each imperfect cache line are diverted to a different perfect cache line. The experimental results are presented in Figure 11. The experiments are conducted for various percentages of affected cache lines; from 15% to 25% to 40%. All the performance values are presented with respect to the corresponding values for the ideal case, where the cache is immune to process variation. The average performance values (harmonic means) across the benchmarks tested for CRBE are 3.0% (2.9%),

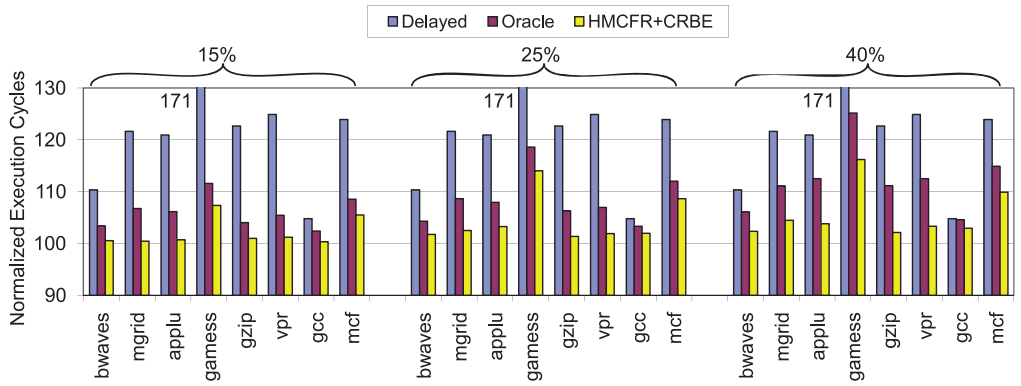

Fig. 12. Normalized execution cycles for various percentages of affected cache lines and TLB entries. The percentages of affected lines and entries vary from 15% to 25% to 40%.

4.2% (4.1%) and 5.7% (5.6%), respectively. The corresponding values for the PADded scheme are 4.1% (4.1%), 8.5% (8.3%) and 13.8% (13.7%), respectively.

As can be seen from the figure, if the percentage of cache lines affected by process variation is small, the performance of the CRBE scheme is generally a little bit better than that of the PADed cache technique. However, as the percentage of affected cache lines increases, our CRBE technique outperforms the PADded technique considerably. This is ascribed to the fact that as the percentage of imperfect cache lines increase, the PAD caches introduce more conflict misses due to disabling imperfect cache lines and re-mapping their references to perfect cache lines. This is especially evident in experiments with the cache having 40% of imperfect lines.

6.3. Evaluating TLB-Oriented and Cache-Oriented Techniques Together

Up to now we evaluated our TLB-oriented and cache-oriented techniques separately. The TLB-oriented techniques were evaluated under the assumption that only the TLB is affected by process variation; the cache-oriented techniques were evaluated under the assumption that only the cache is affected by process variation. A separate evaluation gives us an idea about how effective the technique is in reducing the performance degradation when only the corresponding hardware component is affected by process variation. But due to two main reasons, we need to evaluate our techniques together. First, in a real system process parameter variations can cause variations not only in access latencies of the TLB/cache but also in access latencies of the cache/TLB. Second, attacking only one component will put the other component on the critical path. For example, when we employ only a cache-oriented technique, delayed TLB accesses will constitute the critical path in instruction fetches. So, the cache-oriented technique alone would not bring any significant benefit in reducing the process variation impact on instruction fetches. This makes it necessary to employ a TLB-oriented technique and a cache-oriented technique together.

In this section, we consider a system where the TLB and the cache are affected by process variation. The performance values for such a system are given in Figure 12. Since we have observed similar trends in the remaining benchmarks, here we give results for only eight benchmarks. We conducted experiments for various percentages of affected cache sets and TLB entries, changing the percentage of affected sets and entries from 15% to 25% to 40%. In these experiments, we assume that, for the Delayed scheme the cache access and the TLB access together in the instruction fetch stage are completed within two cycles. For the Oracle scheme, an imperfect entry access in the

cache or in the TLB extends the fetch stage by one cycle, thus, that stage completes in two cycles. The HMCFR+CRBE bar indicates that the hardware-managed CFR is used in address translations and the CRBE scheme is applied when encoding cache access latencies. Here we assume that if the translation does not exist in the CFR register or if the cache access is done in a cache block in an imperfect set, the instruction fetch takes two cycles to complete. As seen in the figure, when the HMCFR and CRBE schemes are applied together, they are very effective at decreasing the varying number of imperfect cache sets and imperfect TLB entries. When we change the percentages of the cache sets and TLB entries affected by process variation from 15% to 25% to 40%, the average performance values (harmonic means) across these eight benchmarks for the Oracle scheme are 6.1% (5.9%), 8.5% (8.3%), and 12.2% (11.9%), respectively, worse than the Perfect scheme. On the other hand, the corresponding performance values for HMCFR+CRBE are only 2.1% (2.0%), 4.4% (4.2%), and 5.6% (5.4%), respectively, deviating less from the performance values of the Perfect scheme. These results tell us that our HMCFR and CRBE schemes are so effective that when they are used together, the performance loss in the instruction fetch stage caused by process variation is more effectively diminished compared to the hypothetical case where the hardware knows in advance whether the cache and TLB accesses would be made in perfect (imperfect) entries, and thus is capable of tuning the timing of the pipeline stages accordingly.

6.4. Evaluating Our Schemes for Different Cache Designs

In this section, we would like to evaluate our TLB-oriented and cache-oriented techniques together for two different instruction cache designs: a nonpipelined cache with 3 access latencies and a pipelined cache with 3 stages. In both setups, the number of any execution unit is one half of the corresponding value given in Table I and the memory latency is 250 cycles. In a nonpipelined cache setup, we assume that an imperfect cache line access may take 4 or 5 cycles, with an equal probability, depending on the severity of variations. In the pipelined cache setup, we take into consideration a 3-stage pipelined cache as suggested in Chishti and Vijaykumar [2004]. 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 the voltage difference of analog bitlines is converted into the digital by sense amplifiers, making the second stage indivisible [Chishti and Vijaykumar 2004]. Process variations 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 experiments we take only the variations into consideration that affect the second pipe stage, delaying the stage by 1 or 2 cycles with an equal probability. More specifically, the perfect cache line accesses complete in 3 cycles while variation affected cache accesses complete in either 4 cycles or 5 cycles. This makes variation affected cache accesses compatible with that of the nonpipelined cache. We have changed the SimpleScalar simulator's code to support the pipelined instruction cache with 3 stages experiencing one or two cycle delays in it's second stage due to variations. In both setups, two unused bit slots in memory access instructions are used to encode three different access latencies. In the pipelined setup, encoded access latencies give a hint to the second pipe stage about how long to wait for imperfect cache line accesses.

The results for both setup is shown in Figure 13. The average performance values (harmonic means) of the nonpipelined cache across the eight benchmarks for the

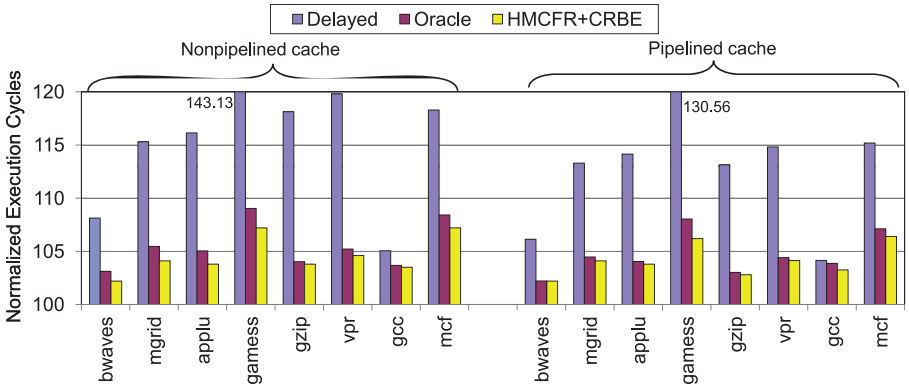

Fig. 13. Normalized execution cycles for the case where 25% of cache sets/TLB entries are assumed to be affected by variations. The left hand side corresponds to non pipelined cache while right hand side corresponds to pipelined cache.

Delayed, Oracle and HMCFR+CRBE schemes are 18.0% (17.1%), 6.5% (6.3%) and 4.5% (4.3%), respectively. The corresponding values of the pipelined cache are 12.0% (11.7%), 4.6% (4.5%) and 4.0% (3.9%), respectively. When we look at Figures 12 and 13, we see that the most variation affected cache form is the one with a single cycle latency and the least affected one is the pipelined cache. From the two figures we can also draw the following conclusion: Although our schemes are best suited for the single cycle latency cache, both the pipelined caches and nonpipelined caches with larger latency, too, can make use of them to lessen the variation introduced performance effects.

7. RELATED WORK

We can broadly classify the current studies on process variations into a circuit level and an architectural level. These studies, in general, use models such as Statistical Static Timing Analysis (SSTA) algorithms [Devgan and Kashyap 2003; Zhan et al. 2005; Sarangi et al. 2007] to analyze the effects of process variations such as circuit delays and excessive leakage, and detect the components that are more susceptible. In circuit level techniques, the vulnerable parts are adjusted by using methods such as gate sizing [Sinha et al. 2005; Raj et al. 2004], adaptive body biasing [Gregg and Chen 2007; Tschanz et al. 2002], and adaptive supply voltages [Chen and Naffziger 2003]. Aside from these common techniques, there are other mechanisms to cope with process variations at the circuit level. For example, by considering the ephemeral lifetime of data cache blocks, on-chip memory architectures based on 3T1D (three-transistor, one-diode) dynamic cells, instead of the classic 6T (six-transistor) static cells, were proposed in cache designs [Liang et al. 2007]. In that study, they investigate a variety of cache refresh and placement schemes to show that the proposed L1 cache architecture can tolerate large process variations with little performance impact compared to 6T SRAM designs.

Architectural-level studies include a variety of schemes and are complementary with circuit level approaches. To improve the cache yield, a process-tolerant cache architecture was proposed [Agarwal et al. 2005a]. In their study, a built-in self test (BIST) circuitry is integrated into the cache structure, which detects faulty cache blocks based on operating conditions. During the runtime cache accesses to a faulty block are directed to a nonfaulty one in the same row, allowing the processor to access the downsized cache with the same address as in the usual way. In Meng and Joseph [2006], the effects of process variation on cache leakage were examined, and

a technique called way prioritization to close subarrays causing high leakage power was proposed. Some techniques to minimize the yield loss due to power and delay violations in the data cache were introduced [Ozdemir et al. 2006]. The number of data cache sets affected by process variation is reduced by rearranging the physical location of cache blocks [Mutyam and Narayanan 2007]. In their study, a cycle-time-stealing approach [Tiwari et al. 2007] is applied to processor pipelines to mitigate the timing effects of software variations on pipeline stages by transferring the time slack of faster stages to slower ones. Variable latency register files and execution units are proposed in Liang and Brooks [2006] to mitigate the timing effect of variations and to improve chip frequency. The reliability optimization of several-process variation mitigating techniques is presented in Fu et al. [2009]. In this study, the adverse reliability effects of variable latency variation techniques are alleviated by reducing the quantity and residency cycles of vulnerable bits; body-biasing techniques are applied to structures by taking the program's reliability characteristics into consideration.

Although the given process variation techniques are effective in general, they may introduce a significant performance penalty (in case of leakage reduction) or increase circuit complexity. Moreover, the literature lacks compiler-based approaches. In this study, we propose some compiler- and architectural-level techniques to handle variations in access latencies across the instruction TLB and instruction cache without undue hardware complexity or performance compromise.

8. CONCLUSIONS

In parallel with scaling VLSI technology, it is becoming increasingly difficult to control transistor quality. As a result, process variation is emerging as an important problem in system design for SRAM-based memory components, such as on-chip caches and TLBs. 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 instruction cache and TLB clearly show that the performance hit resulting from this scheme is intolerable in an instruction fetch stage where address translation and cache access take place. In this study, we first proposed some hardware and software enhancements and then studied various techniques based on those enhancements to mitigate the impact of process variation on the instruction fetch stage. Our solution to the TLB performance problem caused by process variation involves inserting a register between the CPU and the TLB, and storing the last virtual-to-physical address translation in it. With our hardware-managed CFR scheme, for any address translation the TLB access and the CFR access are started in parallel. However, if the translation is found in the CFR register, the TLB access is canceled. Since the execution flow leaves the boundary of the current page very infrequently, the address translation is likely to be obtained from the CFR, and thereby avoiding TLB access and thus also process variation. As to the performance loss due to accessing a process-variation-affected cache, we try to annotate the access latency of the set of instructions to be accessed next in the preceding instruction(s). This latency dictates how many cycles to wait for the corresponding cache block to appear in the output buffer. We proposed several techniques based on the concept of annotation. Our experimental results revealed that our hardware and software techniques are indeed very effective in reducing the overhead caused by process variation. For example, using HMCFR and CRBE (our most effective schemes) together, it is possible to diminish performance loss in the instruction fetch stage more than in the case where the hardware has prior knowledge of whether the cache and TLB accesses would be made in perfect (imperfect) entries and is thus able to adjust the timing of the pipeline stages accordingly.

REFERENCES

AGARWAL, A., BLAAUW, D., AND ZOLOTOV, V. 2002. Statistical timing analysis for intra-die process variations with spatial correlations. In *Proceedings of the International Conference on Computer Aided Design*. 900–907.

AGARWAL, A., PAUL, B. C., MAHMOODI, H., DATTA, A., AND ROY, K. 2005a. A process-tolerant cache architecture for improved yield in nanoscale technologies. *IEEE Trans. VLSI Syst. 13*, 1, 27–38.

AGARWAL, A., PAUL, B. C., MUKHOPADHYAY, S., AND ROY, K. 2005b. Process variation in embedded memories: Failure analysis and variation aware architecture. *IEEE J. Solid-State Circuits 40*, 9, 1804–1814.

ALAM, M. D. 2008. Reliability- and process-variation aware design of integrated circuits. *Microelectron. Reliab. 48*, 8, 1114–1122.

BORKAR, S., KARNIK, T., NARENDRA, S., TSCHANZ, J., KESHAVARZI, A., AND DE, V. 2003. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the 40th Annual Design Automation Conference*. 338–342.

BOWMAN, K., DUVALL, S. G., AND MEINDL, J. D. 2002. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE J. Solid-State Circuits 37*, 2, 183–190.

CHANG, H. AND SAPATNEKAR, S. S. 2005. Full-chip analysis of leakage power under process variations, including spatial correlations. In *Proceedings of the 42nd Annual Design Automation Conference*. 523–528.

CHEN, Q., MAHMOODI, H., BHUNIA, S., AND ROY, K. 2005. Modeling and testing of SRAM for new failure mechanisms due to process variations in nanoscale CMOS. In *Proceedings of the 23rd IEEE VLSI Test Symposium*. 292–297.

CHEN, T. AND NAFFZIGER, S. 2003. Comparison of adaptive body bias (ABB) and adaptive supply voltage (ASV) for improving delay and leakage under the presence of process variation. *IEEE Trans. VLSI Syst. 11*, 5, 888–899.

CHISHTI, Z. AND VIJAYKUMAR, T. N. 2004. Wire delay is not a problem for SMT (in the near future). In *Proceedings of the International Symposium on Computer Architecture*. 40–51.

CHIUEH, T. C. AND KATZ, R. H. 1992. Eliminating the address translation bottleneck for physical address cache. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 137–148.

DEVGAN, A. AND KASHYAP, C. 2003. Block-based static timing analysis with uncertainty. In *Proceedings of the International Conference on Computer-Aided Design*. 607–614.

FU, X., LI, T., AND FORTES, J. A. B. 2009. Soft error vulnerability aware process variation mitigation. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 93–104.

GREGG, J. AND CHEN, T. 2007. Post silicon power/performance optimization in the presence of process variation using individual well-adaptive body biasing. *IEEE Trans. VLSI Syst. 15*, 3, 366–376.

HAMERLY, G., PERELMAN, E., LAU, J., AND CALDER, B. 2005. Simpoint 3.0: faster and more flexible program analysis. *J. Instruction-Level Parallelism 7*, 1–28.

KADAYIF, I., NATH, P., KANDEMIR, M., AND SIVASUBRAMANIAM, A. 2007. Reducing data TLB power via compiler-directed address generation. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. 26*, 2, 312–324.

KADAYIF, I., SIVASUBRAMANIAM, A., KANDEMIR, M., KANDIRAJU, G., AND CHEN, G. 2002. Generating physical adresses directly for saving instruction tlb energy. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*. 185–196.

KNIGHT, J. AND ROSENFELD, P. 1984. Segmented virtual to real translation assist. *IBM Tech. Disclosure Bull. 27*, 2, 1077–1078.

LIANG, X. AND BROOKS, D. 2006. Mitigating the impact of process variations on processor register files and execution units. In *Proceedings of the International Symposium on Microarchitecture*. 504–514.

LIANG, X., CANAL, R., WEI, G. Y., AND BROOKS, D. 2007. Process variation tolerant 3T1D-based cache architectures. In *Proceedings of the International Symposium on Microarchitecture*. 15–26.

MADDOCK, R., MARKS, B., MINSHULL, J., AND PINNEL, M. 1981. Hardware address resolution for variable length segments. *IBM Tech. Disclosure Bull. 23*, 11, 5186–5187.

MANNE, S., KLAUSER, A., AND GRUNWALD, D. 1998. Pipeline gating: speculation control for energy reduction. In *Proceedings of the International Symposium on Computer Architecture*. 132–141.

MENG, K. AND JOSEPH, R. 2006. Process variation aware cache leakage management. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 262–267.

MUTYAM, M. AND NARAYANAN, V. 2007. Working with process varation aware caches. In *Proceedings of the Design, Automation and Test in Europe Conference*. 1–6.

NASSIF, S. R. 2001. Modeling and analysis of manufacturing variations. In *Proceedings of the IEEE Conference on Custom Integrated Circuits*. 223–228.

OZDEMIR, S., SINHA, D., MEMIK, G., ADAMS, J., AND ZHOU, H. 2006. Yield-aware cache architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 15–25.

PAPANIKOLAOU, A., LOBMAIER, F., WANG, H., MIRANDA, M., AND CATTHOOR, F. 2005. A system-level methodology for fully compensating process variability impact of memory organizations in periodic applicatios. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 117–122.

PERELMAN, E., HAMERLY, G., AND CALDER, B. 2003. Picking statistically valid and early simulation points. In *Proceedings of the International Conference on Parallel Architectures and Compiling Techniques*. 244–255.

RAJ, S., VRUDHULA, S. B. K., AND WANG, J. 2004. A methodology to improve timing yield in the presence of process variations. In *Proceedings of the Design Automation Conference*. 448–453.

SARANGI, S. R., GRESKAMP, B., AND TORRELLAS, J. 2007. A model for timing errors in processors with parameter variation. In *Proceedings of the International Symposium on Quality Electronic Design*. 647–654.

SHERWOOD, T., PERELMAN, E., AND CALDER, B. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 3–14.

SHIRVANI, P. P. AND MCCLUSKEY, E. J. 1999. PADded cache: a new fault-tolerance technique for cache memories. In *Proceedings of the 17th IEEE VLSI Test Symposium*. 440–445.

SIMPLESCALAR 2012. SimpleScalar toolset. http://www.simplescalar.com.

SINHA, D., SHENOY, N. V., AND ZHOU, H. 2005. Statistical gate sizing for timing yield optimization. In *Proceedings of the International Conference on Computer-Aided Design*. 1037–1041.

SPEC 2012. SPEC2000 and SPEC2006 Benchmark Suites. http://www.spec.org.

STRECKER, W. D. 1978. VAX-11/780: a virtual address extension to the DEC PDP-11 family. In *Proceedings of the American Federation of Information Processing Societies National Computer Conference*. 967–980.

TANG, X., DE, V., AND MEINDL, J. 1997. Intrinsic mosfet parameter fluctuations due to random dopant placement. *IEEE Trans. VLSI Syst. 5*, 4, 369–376.

TIWARI, A., SARANGI, S. R., AND TORRELLAS, J. 2007. Recycle: pipeline adaptation to tolerate process variation. In *Proceedings of the International Symposium on Computer Architecture*. 323–334.

TSCHANZ, J., KAO, J. T., NARENDRA, S. G., NAIR, R., ANTONIADIS, D. A., CHANDRAKASAN, A. P., AND DE, V. 2002. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *IEEE J. Solid-State Circuits 37*, 11, 1396–1402.

ZHAN, Y., STROJWAS, A. J., LI, X., PILEGGI, L. T., NEWMARK, D., AND SHARMA, M. 2005. Correlation-aware statistical timing analysis with non-Gaussian delay distribution. In *Proceedings of the Design Automation Conference*. 77–82.

ZUCHOWSKI, P. S., HABITZ, P. A., HAYES, J. D., AND OPPOLD, J. H. 2005. Process and environmental variation impacts on ASIC timing. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*. 336–342.

Received November 2012; revised March 2013; accepted May 2013