Appears in the ACM International Conference on Computing Frontiers
Ischia, Italy, May, 2008

Reducing the Impact of Intra-Core Process Variability with Criticality-Based Resource Allocation and Prefetching

Bogdan F. Romanescu, Michael E. Bauer, Sule Ozev, and Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University
P.O. Box 90291
Durham, NC 27708, USA
{bfr2, meb26, sule, sorin}@ee.duke.edu

ABSTRACT

We develop architectural techniques for mitigating the impact of process variability. Our techniques hide the performance effects of slow components—including registers, functional units, and L1I and L1D cache frames—without slowing the clock frequency or pessimistically assuming that all components are slow. Using ideas previously developed for other purposes—criticality-based allocation of resources, prefetching, and prefetch buffering—we allow design engineers to aggressively set the clock frequency without worrying about the subset of components that cannot meet this frequency. Our techniques outperform speed binning, because clock frequency benefits outweigh slight losses in IPC.

Categories and Subject Descriptors

C.1.0 Processor Architectures, C.4 Performance of Systems

General Terms

performance, reliability

Keywords

process variability, microarchitecture

1. INTRODUCTION

A major problem facing the computer and semiconductor industries is the increasing amount of CMOS process variability [3, 9]. As transistor and wire dimensions continue to shrink, the variability in these dimensions has a greater impact [17]. Variability in low-level circuit parameters, such as transistor gate length and gate oxide thickness, complicates system design by introducing uncertainty about how a fabricated system will perform. Although a circuit or chip is designed to run at a nominal clock frequency, the fabricated implementation may vary far from this expected performance.

There are useful transistor-level and circuit-level techniques, such as adaptive body biasing (ABB) [23], gate sizing, Razor flip-flops [7], and X-Pipe flip-flops [24], that can help to mitigate the impact of variability, but they cannot solve the problem entirely. For example, Razor flip-flops cannot replace functional flip-flops which are at the ends of either critical paths (paths that typically determine the clock frequency) or short paths whose delays are less than half of a cycle. Thus, we will still have some components (e.g., functional units) that are slower than other identical components. Our goal in this work is to allow the processor to be clocked faster than its slowest components.

In this paper, we develop architectural techniques for mitigating the impact of process variability on three performance-critical microprocessor components. We make the following three contributions:

- We alleviate the impact of having slow access latencies for some registers, by renaming the destinations of critical instructions to fast registers.
- We reduce the impact of slow functional units by giving critical instructions priority for the fast functional units.
- We mitigate the impact of having some L1 cache frames that are slower than others, by prefetching into small prefetch buffers.

All three of our approaches enable us to aggressively clock the processor, and thus increase the fraction of slow components, without significantly degrading the IPC (instructions per cycle). For example, we can clock the processor so fast that 10% of its L1I cache frames cannot be accessed within the clock period, yet we incur only a slight IPC loss. The benefit in clock frequency outweighs the loss in IPC, providing us with a net gain in performance.

Our work combines several ideas—criticality, prefetching, and prefetch buffering—that were previously developed for other purposes. *Our contributions are using and combining these ideas to overcome the effects of process variability.*

We discuss process variability, its potential impact, and how we model it in Section 2. We describe our experimental meth-

CF'08, May 5-7, 2008, Ischia, Italy.

odology in Section 3. In Sections 4 through 6, we present our three variability tolerance schemes. We compare this work with related work in Section 7, and we conclude in Section 8.

2. PROCESS VARIABILITY

In this section, we discuss process variability in more detail, in order to motivate our research. Process variability arises due to several specific causes, but the over-arching cause is the inability to perform VLSI fabrication with every feature *exactly* as planned. The design might specify that a transistor is 45nm long, but, due to fabrication imperfections, some transistors may be somewhat shorter or longer.

2.1 Impact of Process Variability

In addition to affecting a transistor's length and width, process variability also has a non-trivial impact on a transistor's gate oxide thickness and threshold voltage. These four low-level parameters—*L, W,* t_{ox}, and V_{t0},—are generally considered to be the most sensitive to process variability [17].

The challenge for architects and circuit designers is that low-level process variability impacts the behavior of transistors and wires, and this impact propagates up to gates, subsystems, and processors. Currently, there are three primary ways of dealing with variability that is not tolerated with low-level circuit techniques, but each of these approaches has drawbacks. We describe each of them in the context of a system with many generic components (e.g., cache frames, functional units, etc.), some of which are slower than others.

- *Speed binning*: Decrease the clock frequency for the whole chip such that the latency of slow components still fits into its nominal number of cycles. Key disadvantage: the whole processor is slowed down to accommodate some slow components.
- *Deconfiguration of slow components*: Never use any components that are deemed too slow and thus do not sacrifice clock frequency. Key disadvantage: deconfiguring slow components reduces the effective number of components and hurts IPC.
- *Pessimistic design*: Assume that the component latency is expected to be *C* cycles in the case without process variability. Design the processor such that all components are expected to take more than *C* cycles. Key disadvantage: all components take one or more extra cycles, even when not strictly necessary. The throughput disadvantage of pessimistic design can be mitigated in some situations by pipelining, but the latency penalty cannot be avoided.

There also exist hybrids of these three approaches. In particular, a designer can specify the latency boundaries between components that are "fast" and "slow", deconfigure the "slow" components, and then speed bin the chips. Nevertheless, any of these approaches or hybrids suffer from problems that we want to overcome.

2.2 Modeling Process Variability

As part of this project, we have used the publicly available VariaSim [19] tool for determining the impact of process variability on circuit performance. VariaSim has enabled us to obtain performance results for specific circuits with specific technology assumptions.

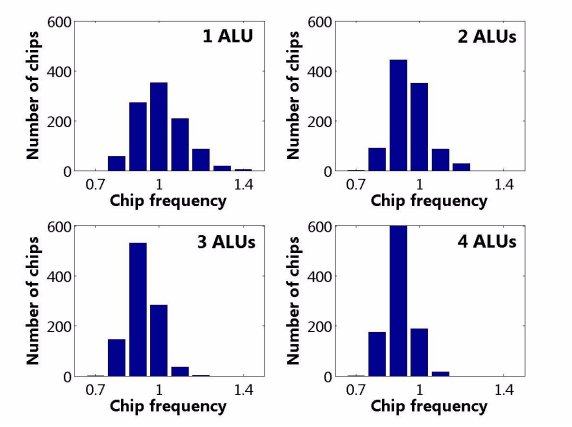

Figure 1. Speed Binning. Chip frequencies are normalized with respect to f_{nom}.

As an illustrative example, we consider the impact of variations in only *L*. Assume that the processor's frequency is determined by the integer execution stage. We generate four sets of 1000 chip configurations, where each set has a fixed number of integer ALUs (borrowed from Sun's open-source Niagara T1 [13]) ranging from 1 to 4. We generate the different chip configurations considering both spatial and random variations in *L*, with a nominal value L_{nom}= 65nm and a total variability of 40% of L_{nom}. We used VariaSim to determine the delays for all of the chips, and then we binned the chips into 8 bins that are equally spaced between 0.7*f_{nom} and 1.4*f_{nom}, where f_{nom} is the frequency of one ALU in the absence of process variability. As shown in Figure 1, increasing the number of ALUs increases the likelihood that one or more will be slower than f_{nom} and thus lead to a slower clock. In this paper, we avoid slowing the clock to satisfy the slowest ALU. For example, for the 4-ALU chips, we can use our techniques to treat the *k* slowest ALUs as being 2-cycle units (i.e., *k*/4 of the units are considered slow). This approach can maintain high clock frequencies with only small degradations in IPC, leading to an overall performance increase compared to just speed binning.

For this paper, we are reluctant to commit ourselves to any assumptions about the technology, especially because our work is independent of these exact details. Rather, we parameterize variability (e.g., *X*% of cache frames or registers are slow), so that we can abstract away the low-level physical phenomena that cause variability. This is a technique previously used in other academic studies, due to the lack of publicly available information from industry. We sweep over the range of possible variabilities (e.g., 5-40% slow cache frames or registers) that are probable for a given technology.

3. EXPERIMENTAL METHODOLOGY

To evaluate our designs, we modified SimpleScalar [2]. For most of our experiments, we study a superscalar processor that is modeled roughly after the Alpha 21264 [12]. Table 1 shows the detailed configuration of the baseline superscalar processor we model. For a few experiments, we study a simpler, in-order

Table 1. Parameters of Superscalar Processor

Feature	Details
pipeline stages	8: fetch, decode, rename, issue, regread, execute, writeback, commit
width: fetch,issue, commit	4, 4int/2fp, 4
branch predictor	Gshare: 8-entry BHR, BHT has 4K 2-bit entries
instruction window	20 int, 15 fp
physical register file	80 int, 72 fp
reorder buffer	80 entries
load/store queue	32 load, 32 store
integer units	4 ALUs (1 cycle), 1 mult (7), 1 divide (7)
floating point units	2 ALUs (4 cycles), 1 mult (4), 1 divide (12), 1 sqrt (24)
L1 I-Cache	32KB, 2-way, 1 cycle
L1 D-Cache	32KB, 4-way, 1 cycle
L2 cache (unified)	2MB, 8-way, 14 cycles
memory	120 cycles

processor. This in-order processor is 2-wide and uses branch prediction. Its parameters are the same as those in Table 1, except it does not have the structures used to coordinate out-of-order execution.

For benchmarks, we use the complete SPEC2000 benchmark suite with the reference input set. To reduce simulation time, we used SimPoint analysis [20] to sample from the execution of each benchmark.

Quantitatively comparing our approaches to speed binning is challenging, because we do not have access to industrial data on the probability distributions of access latencies, but we can explore several Gaussian distributions with various standard deviations. After cutting off the slowest *X*% at the tail end of the distribution (i.e., treating them as "slow"), we determine how fast we can clock the processor. We compare this result to the 99.5% point (worst case), and we consider the remaining 0.5% defective and assume they will be discarded during production testing. As the standard deviation increases, the differential between the (100-*X*)% point and the 99.5% point increases, favoring our approaches. Compared to speed binning, our approaches may have slightly lower IPC, but our clock frequency is greater.

4. REGISTER FILE

Due to process variability, we assume that accesses to some physical registers take more or less time than accesses to other physical registers. We assume that production testing or built-in self-test (BIST) can determine the access latencies of each register and store it for future reference. We set the clock frequency such that the majority (e.g., 80%) of the registers can be accessed within one cycle. The remaining registers are "slow" and take one additional cycle to access.

Existing options for dealing with variability in register file access latency all have significant drawbacks. Speed binning can slow the entire processor, and deconfiguration of slow registers reduces the effective size of the register file. Pessimistic design leads to a 2-cycle access latency for all registers, which has several negative implications. First, when register reads are on the critical path, performance will be degraded. Second, the extra latency of each read and write will double the contention for the register file and force the implementation to either pipeline the register file access or add ports.

Recently, Liang and Brooks [16] developed a scheme for steering register reads to faster port/register combinations. They add multiplexors and logic to enable an instruction to read both of its register operands from fast port/register combinations when possible. Unlike their work, we assume that latency is a function of the register (not the port/register pair), and we address the issue of register writes, which they explicitly left for future work. We also avoid the use of multiplexors on the pipeline's critical path.

Another option is to pipeline the register file, but this solution is more difficult to implement and expensive than our approach. Pipelining also does not help register read latency.

4.1 Criticality-Based Register Allocation

Our approach, *criticality-based register allocation (CRA)*, tries to steer critical instructions to the fast (1-cycle) registers. Because the performance of non-critical instructions has little or no effect on program performance, we want critical instructions to have priority for the fast registers.

To identify critical instructions, we borrow the previously developed instruction criticality predictor from Fields et al. [8]. This predictor has as many entries as the ROB (80), and each entry is 24 bits. Because of its small size, its access latency is short—even in the presence of significant process variability—and can be overlapped with the first stage or two of the front end.

When a critical instruction reaches the Rename stage, its destination register is renamed to a free fast register, if one is available. CRA does not ensure that critical instructions will *read* from fast registers, because we do not control which registers are read. However, by having a critical instruction, *A*, write to a fast register, we enable instructions dependent on *A* to read *A*'s output from a fast register or the operand bypass network. Because *A* is critical, it is likely that at least one instruction that reads its output is also critical. If one of these critical dependent instructions must read *A*'s result from the register file (instead of from the bypass network), then CRA succeeds.

We implement CRA by separating the "Free Register List" into two separate lists, one for fast registers and one for slow registers, as illustrated in Figure 2. When deallocating a register, the processor looks up whether it is a fast or slow register in order to modify the appropriate free list. CRA must tolerate the fact that register accesses, both reads and writes, may take either one cycle or two cycles.

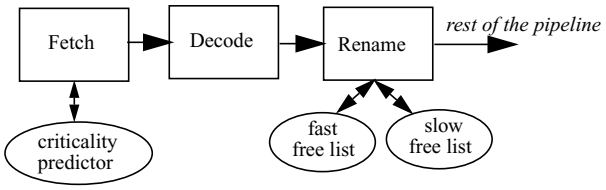

Figure 2. CRA: Register Allocation

4.1.1 Reads

Like Liang and Brooks [16], when an instruction reads one or two slow input registers, the instruction is stalled for an additional cycle before it can execute. This stall propagates backwards to prevent other instructions from trying to use the busy read port(s). If we are frequently accessing slow registers, which could happen if there is a lot of pressure on the register file and many reads are to slow registers, then this added occupancy will increase contention for read ports. We may have to add read ports if we expect this situation to arise frequently, but ports incur significant area and delay costs.

4.1.2 Writes

CRA incurs fewer slow (2-cycle) writes than an oblivious approach that treats all registers equally, because the renaming of destination registers is prioritized to choose fast registers. However, the possibility of *any* slow writes presents a correctness issue. Consider the latch between the output of a 1-cycle functional unit and the register write port. If the write takes two cycles, then the ALU could potentially overwrite the latched value before it completes writing to the register file.

We present three solutions to this problem, in increasing order of performance and complexity.

Always Pessimistic. An overly simplistic solution is to have the scheduler always treat the functional unit as a 2-cycle component instead of 1-cycle. This solution incurs a significant performance penalty.

Stall Functional Unit if Slow Write. The scheduler keeps track of when a functional unit will be writing to a slow register (for instruction *A*) and then does not schedule that functional unit (for some later instruction in program order, say *A+1,* although in a superscalar processor it could be some other instruction) for an additional cycle. This improved solution still potentially hurts performance by preventing the functional unit from producing *A+1*'s result, which could be consumed by its dependent instructions (even if a write port is not yet available). For our target system, performance degradation due to this effect is minimal, so we use this fairly simple approach in our experiments.

4.2 Evaluation

The goal of these experiments is to determine the potential of CRA to mitigate the impact of process variability in the register file. We provide enough read and write ports to avoid common-case contention (i.e., in the absence of variability), but there can be contention due to slow reads and writes.

We compare CRA performance to four other options: speed binning, pessimistic design (assuming all register accesses take two cycles), deconfiguration (not using slow registers), and simply using the slow registers but without CRA's allocation policy. We do not quantitatively compare to Liang and Brooks, due to our different assumptions about how variability manifests itself and because they do not consider variability in write access latencies.

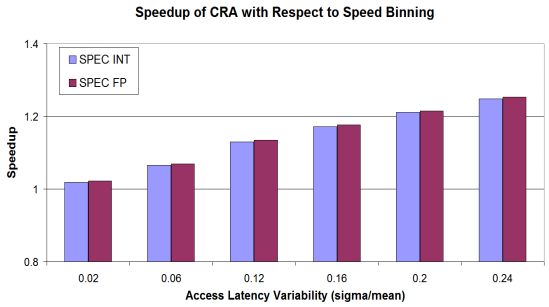

Figure 3. Speedup of CRA with respect to speed binning

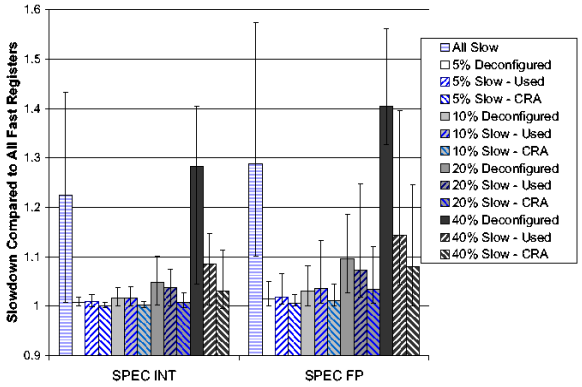

Figure 4. CRA Performance. Error bars represent the min/max across the benchmarks.The min bars do not include mcf, because its value is near zero and obscures the other data.

In Figure 3, we compare the performance of CRA to simple speed binning, using the methodology explained in Section 3. For CRA, we allow the slowest 20% of the registers to be "slow". We average across the SpecINT and SpecFP benchmarks in order to present a tractable amount of data. The x-axis represents the amount of variability (sigma/mean) in register access latency[1], and the y-axis is CRA's speedup (in units of instructions per *second*) with respect to speed binning. As variability increases, CRA achieves increasing speedups, because it does not need to slow its clock down appreciably and it only incurs slight IPC degradation. Speed binning maintains a constant IPC but sacrifices clock frequency.

In Figure 4, we compare CRA to the other three options. The results are presented as *slowdowns* with respect to an ideal processor with no process variability (i.e., all registers are fast). The results reveal that pessimistic design performs poorly, but that the *average* slowdowns for the other three approaches are modest until 20% of the registers are slow. CRA is somewhat better, on average. However, the min/max bars reveal that CRA's worst-case is far better than the other

1. Sigma/mean is the coefficient of variation.

options. Furthermore, as the fraction of slow registers increases, both CRA's average and worst-case become increasingly favorable as compared to the other options.

5. FUNCTIONAL UNITS

Due to variability, it is likely that some functional units (FUs) will operate more slowly than others. These FUs include integer and floating point adders, multipliers, and dividers. Our goals are similar to those for the register file. We want to avoid slowing down the clock to the entire chip to accommodate a slow FU. As with CRA, we also want to avoid deconfiguring (mapping out and never using) a slow FU, because it is still useful. We assume that we can speed test the FUs during production test (BIST is generally only used for memory structures) and upload this information into the processor.

5.1 Criticality-Based FU Allocation

We address this problem by allowing a functional unit that is substantially slower than other identical FUs to take one or more additional cycles. For example, a nominally 1-cycle adder could be extended to 2 cycles. A 7-cycle fully-pipelined multiplier could be extended to somewhere between 8 and 14 cycles, depending on how many of its stages could be slow. For example, if any of the 7 stages could be slow, then a simple solution is to allow every stage to take two cycles, thus extending the total latency to 14 cycles

To allow variable latency FUs, we must address two issues. First, we must avoid putting slow FUs on the critical path of a program's execution. Second, the instruction scheduler must accommodate variable latencies.

Our scheme is *criticality-based functional unit allocation (CFUA)*. We make a small modification to the instruction scheduling logic (refer to Brown et al.'s baseline scheduler as an example [4]) to enable it to handle two possible cycle latencies for each FU. Schedulers can already handle FUs with different cycle latencies, such as 1-cycle ALUs and multi-cycle multipliers and floating point units. The difference for CFUA is that a FU may take either its nominal number of cycles, *C*, or more cycles, and the scheduler will not know this until after the chip has been fabricated and tested. Instead of hardwiring into the scheduler that a given FU takes *C* cycles, we use a multiplexor to choose between *C* and the slow scenario (which is between *C+1* and *2C* cycles), depending on the results of the production test. Thus, the scheduler will wait the correct number of cycles before scheduling dependent instructions.

CFUA does not affect the operand bypass network, because we introduce no new bypass paths. The output of a slow FU can still be forwarded to the inputs of the other FUs (as before), using the same wires and muxes. Our proposed changes focus on the scheduler itself and do not add complexity to the forwarding logic.

5.2 Evaluation

We compare the performance of CFUA to speed binning, pessimistic design (all FUs are slow), allowing some FUs to be slow but without criticality-based allocation, and deconfiguration of slow FUs. We vary the number of ALUs and FPUs that

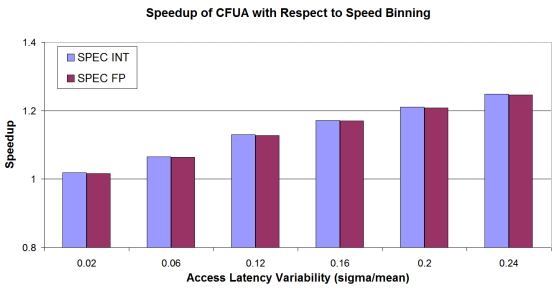

Figure 5. Speedup of CFUA with respect to speed binning

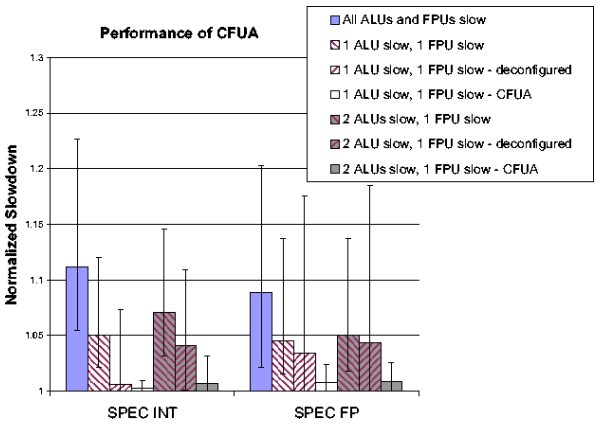

Figure 6. CFUA Performance. Error bars represent the min/max across the benchmarks.

are slow. We do not consider the multipliers or dividers, because their performance is generally not critical. We assume full pipelining of the FUs, and we assume that slow FUs take only *C+1* cycles, which is the minimum extra latency and favors pessimistic design (rather than CFUA).

Figure 5 shows CFUA's speedup with respect to speed binning. In this experiment, we assume that two ALUs are slow and one FPU is slow. We observe that CFUA consistently outperforms speed binning, with the benefit increasing as the amount of variability increases.

In Figure 6, we compare CFUA to the other techniques, and the results are shown as slowdowns with respect to an ideal processor with no process variability (i.e., all FUs are fast). The results reveal that CFUA outperforms the other options, especially when considering the worst-case benchmarks. CFUA thus enables aggressive clocking by hiding the impact of the slow FUs. Another observation is that deconfiguring slow FUs is often preferable to using them, because critical computations will always use fast FUs (if they are available).

6. L1I AND L1D CACHES

The last structures that we address in this work are the L1I and L1D caches. We assume that, due to process variability, the access times for different cache frames will differ. This assumption, which is the same as in most of the literature (e.g., Agarwal et al. [1]), reflects variability in storage cells, word decoders, and word lines. Variability in a bit line or sense amp

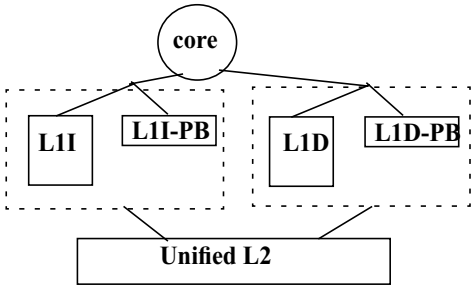

Figure 7. SFP Cache Hierarchy

will manifest as a slow column (i.e., a slow bit in all frames) within a given bank of the L1. One or more slow bits in a frame causes that frame to be slow. Because of three scenarios—slow columns, spatially correlated variability, and variability in the decoders that choose which cache set to access—we also consider the situation in which slow frames are clustered (rather than randomly spread). We assume the processor can determine which frames are slow during production testing (or BIST) and store this information for future reference.

We now describe a method to aggressively clock the processor while not forcing all L1 accesses to pessimistically take one or more extra cycles. We deconfigure slow frames (in Section 6.5, we discuss a more complicated design that keeps the slow frames), but we overcome this loss of frames by prefetching into very small L1I and L1D prefetch buffers (e.g., 2-16 frames, similar in size to a victim cache [11]) that have only fast frames. We can guarantee that an L1 prefetch buffer (L1-PB) has only fast frames, despite variability, by deconfiguring slow frames in the L1-PB (e.g., build with 10 frames and deconfigure the slowest 2). By using only simple prefetchers, we can also guarantee that they are fast, even in the presence of variability. Because we are trying to prefetch blocks into an L1-PB that could otherwise end up in slow L1 frames, we refer to our scheme as *Slow Frame Prefetching (SFP)*. We illustrate the SFP cache hierarchy in Figure 7.

SFP combines two well-known ideas, prefetching and prefetch buffers, that were previously developed for other purposes. Prefetching hides memory access latency, and we leverage existing prefetchers to avoid adding extra variables into our experiments. Prefetch buffers have been used to prevent polluting the cache with unnecessarily prefetched blocks. *SFP's contribution is not the prefetcher or the prefetch buffer; rather, it is the idea to prefetch blocks into the prefetch buffer in order to avoid L1 misses that would occur because of having deconfigured slow L1 frames.*

6.1 Operation

We specify the SFP cache hierarchy operation in Table 2, for an L1 and L1-PB that both use LRU replacement. The L1 and L1-PB are accessed in parallel in one cycle, and it is only possible to hit in one or the other or miss in both, but an access cannot hit in both the L1 and L1-PB. If all blocks in an L1 set are deconfigured, an access will either hit in the L1-PB or be forced to access the L2 (bypassing the L1). A miss in both the L1 and L1-PB has a performance penalty equal to that of an L1 miss in a system without an L1-PB.

Table 2. SFP operation for loads/stores

L1	L1-PB	actions
hit	miss	1-cycle L1 access; update L1's LRU bits
miss	hit	1-cycle L1-PB access; update L1-PB's LRU bits; move LRU slow frame in L1 to MRU

miss	miss	access L2;
		if (LRU frame in L1 is fast)
		  fill it;
		else
		  fill PB, update L1-PB's LRU bits;
		move LRU frame in L1 to be MRU;

hit	hit	*impossible*

There are two aspects of SFP operation that are worthy of further discussion. First, when an access misses in both the L1 and the L1-PB, SFP must decide where to put the block when it arrives from the L2. If the least-recently-used frame in the L1 is fast, then we fill it, just like in a non-SFP system. Otherwise, we place the block in the L1-PB, even though this fill is not actually due to a prefetch.

The second unusual aspect of SFP cache operation is the policy for updating the L1's LRU bits. We maintain the LRU bits for *all* frames in an L1 cache set, not just the fast frames. The situation that differs from non-SFP systems is an L1-PB hit. In this situation, the L1's LRU frame is moved to the most-recently-used position. This policy ensures that, in a cache set with at least one fast and one slow frame, a block in a fast frame will eventually become least-recently-used and then get replaced. Intuitively, for purposes of updating the L1's LRU bits, we pretend that we are actually using the slow L1 frames. We assume that a hit in the L1-PB is like an L1 miss that caused the L1's LRU frame to be replaced in favor of the newly accessed block. This assumption is not always correct—for example, an L1-PB hit could have been to the same block as the previous access—but it works well and there are no correctness issues for LRU bit updates.

6.2 Prefetch Buffer Structure

We assume that the L1I-PB and L1D-PB are fully-associative with LRU replacement. The L1-PBs are write-allocate, and they must participate in cache coherence. It is important to note that the number of frames in an L1-PB can be significantly less than the number of slow frames in the L1 and still provide a large benefit. The key is that programs generally do not touch all the slow frames in the L1 at the same time.

6.3 L1 Instruction Cache Specifics

L1I cache performance is critical to microprocessor performance. L1I caches are almost always either 1-cycle or pipelined such that an access can be initiated every cycle.

We use a simple next-block prefetcher to bring blocks into the L1I-PB. The choice of prefetcher is independent of SFP, but a better prefetcher would lead to better results for SFP.

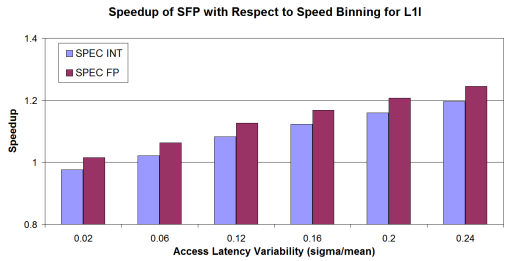

Figure 8. L1I Cache: Comparing SFP to Speed Binning

6.4 L1 Data Cache Specifics

The criticality of L1D cache latency depends on the processor model. For dynamically scheduled (out-of-order) superscalar processors, the latency of non-critical L1D hits can often be at least partially hidden while the processor works on other instructions [21]. However, for in-order cores, such as Niagara T1 [13], the L1D latency is on the critical path. It is likely that future multicore processors will have a mix of in-order and out-of-order cores, in order to not exceed power constraints, and thus the latency of the L1D is important.

To prefetch into the L1D-PB, we use a simple stride prefetcher. A more sophisticated prefetcher, such as a Markov prefetcher [10] or dead block prefetcher [14], would likely improve SFP's results.

6.5 Alternative Design Points

We now discuss three other design points.

Full Pipelining. It might appear that pipelining the L1 would make pessimistic design (adding one or more cycles to each access) an almost penalty-free design option. With pipelining, the latency of each L1 access increases, but the core can still initiate one access per cache port per cycle. This latency penalty is not too problematic, because extra L1I cycles just increase the branch misprediction penalty, and extra L1D cycles can often be hidden by the out-of-order core [21]. However, a problem arises because caches cannot be pipelined arbitrarily finely; one cannot latch the weak bitline signals before they are amplified by the sense amps [5]. To achieve fast clock frequencies, architects break the L1 into sub-arrays such that the delay for the bitlines and sense amps just fits within the desired clock period. A pessimistic design approach would thus have to allocate two cycles for this delay, to accommodate the possibility of delay variability. This delay not only affects L1 access latency, which is not too painful, but also affects the more important L1 throughput. One could sub-bank the L1 even more finely, but increased sub-banking eventually increases access latency.

Using Spare Frames. For fault tolerance, some caches have a small number of spare frames. One could imagine using these spares like SFP uses the L1-PB. The key problem with using spares, though, is that it requires a level of indirection on the critical path of all accesses.

SFP without Deconfiguration of Slow Frames. One could imagine implementing SFP without deconfiguring the slow L1 frames. In such a scheme, an L1-PB miss that hits in a slow L1 frame would avoid the greater penalty of an L1 miss. However, there are two problems with this approach. First, handling slow tag array accesses is difficult, because we do not know if the access is a slow hit before we potentially want to start a subsequent access. We could use testing to identify slow tags and keep this information in a table, but we still would not know about tag matches. Second, if the cache is pipelined, testing will not reveal which of the stages is slow, so we would pessimistically have to assume that all of them are slow. Our results suggest that the added complexity of solving these problems is not worth the effort; however, for future system configurations, we may wish to re-visit this design option.

6.6 Evaluation

In this section, we evaluate the performance benefit of SFP. There are many variables to consider, but we focus only on variables that are particularly relevant to SFP. Thus we fix the cache sizes (32KB) and cache block size (32 bytes). The variables we explore are: L1-PB cache size, L1 cache set-associativity (always using LRU replacement), the fraction of L1 cache frames that are slow, and whether slow L1 cache frames are spatially clustered. Another potentially interesting variable is the nominal number of cycles for an L1 access latency, C, but space constraints preclude analyzing sensitivity to this parameter. We fix C at 1-cycle for both the L1I and L1D.

We compare SFP to three options: speed binning, deconfiguration of slow L1 frames but without an L1-PB (similar to Agarwal et al. [1]), and pessimistic design. *All design options and the baseline (all fast frames) use the same prefetcher*, in order to balance the comparison; the non-SFP schemes prefetch into the L1, whereas SFP prefetches into the L1-PB. For pessimistic design, all accesses take two cycles with the pipeline depth limited by the bitline and sense amp delay. Groups of fetched instructions are available every two cycles.

6.6.1 L1I Cache

In our first experiments, we study a baseline configuration in which the L1I is 2-way set-associative and the L1I-PB has 8 frames. For now, we fix the fraction of L1I frames that are slow at 20% and we assume that slow entries are randomly distributed throughout the L1I.

We plot the speedup of SFP for the L1I, with respect to speed binning, in Figure 8. The results show that for very small amounts of variability, speed binning is actually slightly preferable. However, as the amount of variability increases, SFP outperforms speed binning by a substantial margin. We will not present speed binning results for the L1D, because the data is quite similar.

In Figure 9, we compare SFP to the other techniques. The figure plots slowdowns with respect to the ideal case (all fast frames), and it reveals that both pessimistic design and deconfiguration can degrade performance by a significant amount. Pessimistic design is usually preferable to deconfiguration, but it still averages slowdowns of 19% and 13% for the integer and floating point benchmarks, respectively. SFP with only an 8-frame L1I-PB achieves slowdowns of only 5% and 1%, respectively. Looking at individual benchmarks, we often observe even greater disparity between SFP and its alternatives. For swim, bzip, and galgel, the best choices of the alter-

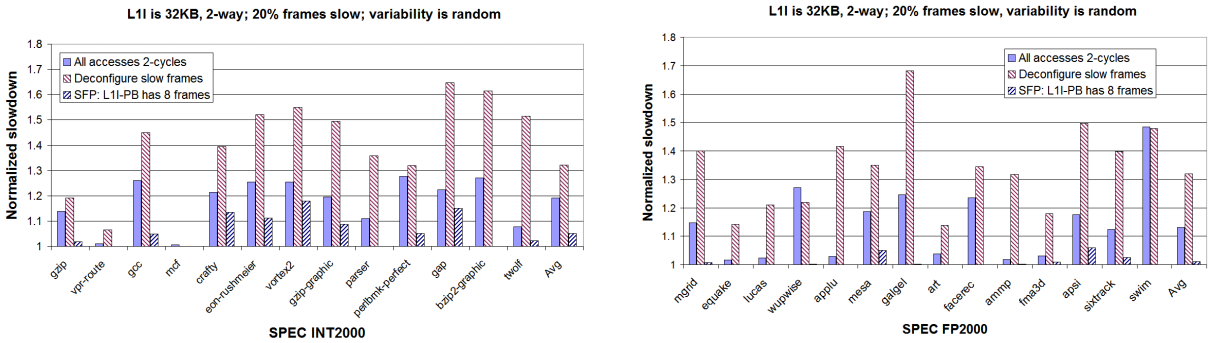

Figure 9. L1I Cache: Comparison of Pessimistic Design, Deconfiguration, and SFP. Slowdown results are with respect to case when all accesses are 1-cycle.

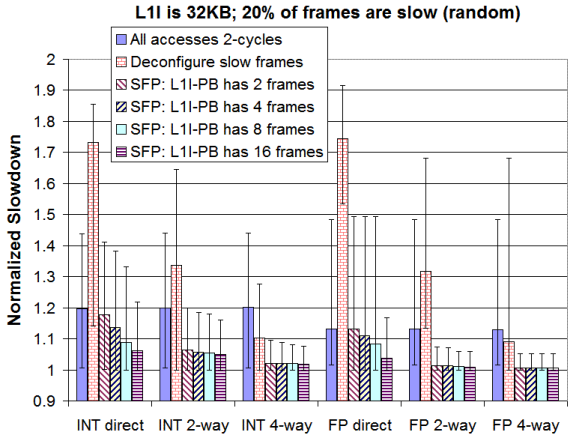

Figure 10. L1I Cache: Sensitivity to L1I set-associativity and L1I-PB size. Slowdown results are with respect to case with all 1-cycle accesses.

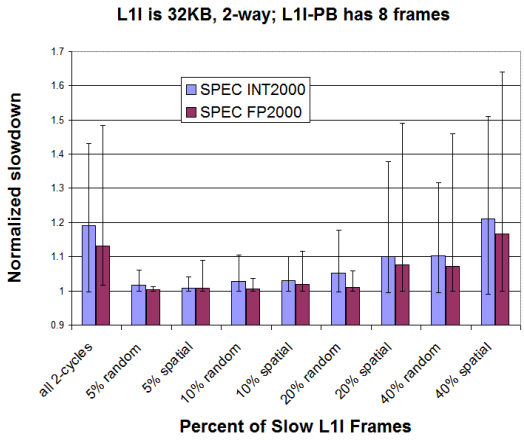

Figure 11. L1I Cache: SFP's Sensitivity to Fraction of Slow Frames. Slowdown results are with respect to case with all 1-cycle accesses.

natives have penalties of 20-45%, whereas SFP has a slowdown less than 1%. These benchmarks are particularly sensitive to slow and deconfigured frames.

In our next experiment, we study the impact of both L1I set-associativity and L1I-PB size. We continue for now to fix the fraction of slow L1I frames at 20% and assume a random distribution of slow frames. In Figure 10, we show the results, averaged across the benchmarks. The "error bars" represent the min/max across the benchmarks. We observe that having only two L1I-PB frames is sufficient except in the case of a direct-mapped L1I. A set-associative cache is less sensitive to having deconfigured frames, because it incurs fewer conflict misses. The need for just a few L1I-PB frames makes SFP a low-cost design option.

In our final L1I experiment, we explore different fractions of slow frames and the effect of spatial correlations. For spatial correlations, we assume that all frames in certain randomly-chosen sets are slow. Figure 11 shows the results for SFP and pessimistic design (for comparison), averaged across the benchmarks, and we observe a few phenomena. First, SFP maintains average slowdowns less than 3% as long as the fraction of slow frames is no more than 10%. Second, SFP is preferable, on average, to pessimistic design in all cases except 40% spatial. Third, when the fraction of slow frames is as high as 20% or 40%, spatial correlations become more problematic for SFP, particularly for the worst-case benchmarks. These results suggest that we can use SFP to aggressively clock the L1I, but probably not beyond the point of 20% slow frames.

6.6.2 L1D Cache

We perform the same set of experiments for the L1D cache as we did for the L1I cache, except we omit the speed binning comparison. Because our experiments confirmed that variability matters far less for out-of-order cores than for in-order (data not shown due to space constraints), we only present results for an in-order core in this section. The results are in Figures 12 through 14.

In Figure 12, we observe that SFP is only slightly better than deconfiguration, on average, for the floating point benchmarks. The biggest advantage is only about 4%, and SFP has some negligibly slight disadvantages (<1%) on two benchmarks. However, for the integer benchmarks, SFP offers substantial gains over the other options. In particular, SFP has a huge advantage for eon, mcf and gcc, and it has a non-trivial advantage for vortex, gap, and bzip. These benchmarks are sensitive to L1D size and lose substantial performance due to deconfigured frames, even with prefetching.

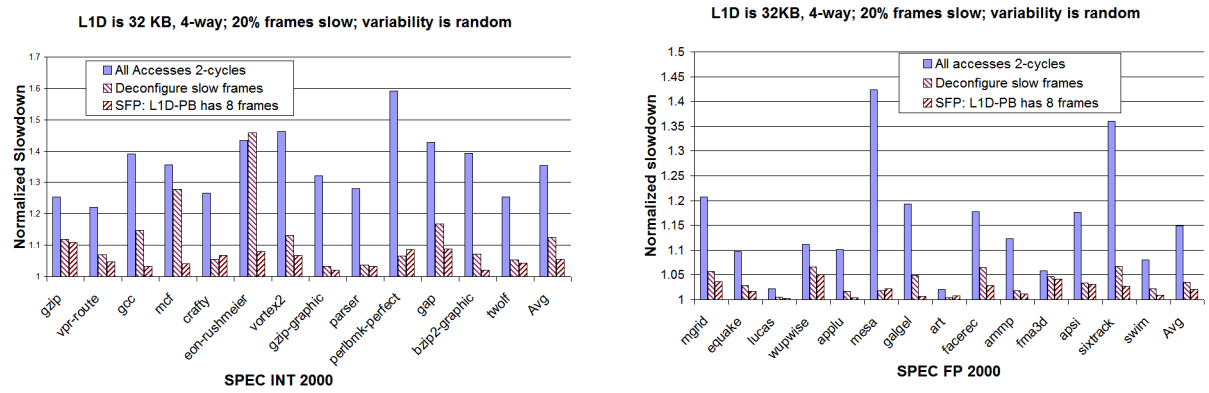

Figure 12. L1D Cache: Comparison of Pessimistic Design, Deconfiguration, and SFP. Slowdown results are with respect to case when all accesses are 1-cycle.

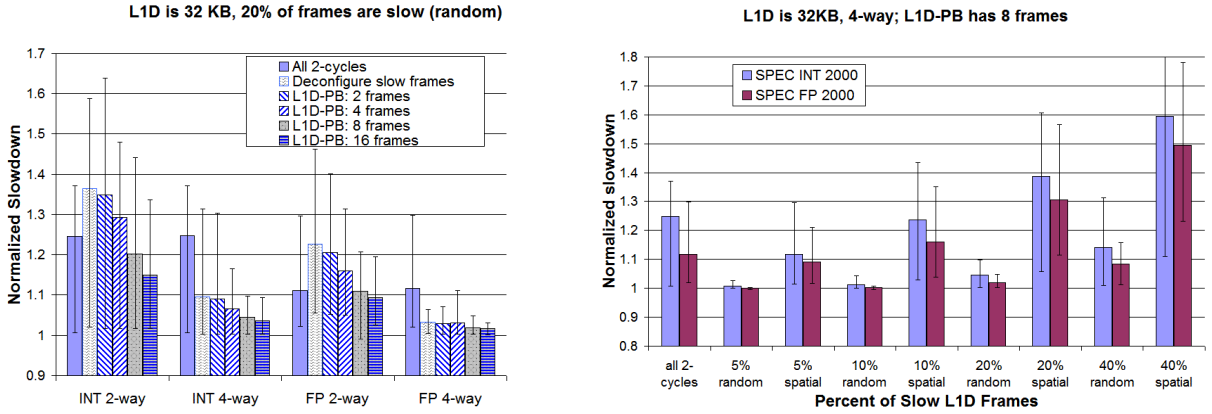

Figure 13. L1D Cache: Sensitivity to L1D set-associativity and L1D-PB size

Figure 14. L1D Cache: Sensitivity to Fraction of Slow Frames

Figure 13 reveals that SFP can, on average, tolerate small L1D-PB sizes. However, for a 2-way L1D, there are some benchmarks for which the L1D-PB needs to have at least 16 frames. The 4-way L1D does not benefit much from having an L1D-PB greater than 8 frames.

Figure 14 shows a much greater impact of spatial correlations than was the case for the L1I. For random variability, SFP incurs little IPC loss even when 20% of the frames are slow. However, spatial correlations are much more problematic, especially for greater than 5% slow.

7. RELATED WORK

There is related work in the areas of mitigating the impact of process variability on performance and timing speculation.

Mitigating Impact on Performance. Agarwal et al. [1] deconfigure slow cache frames. Ozdemir et al. [18] deconfigure slow portions of the L1 cache to improve yield, and they also consider letting some accesses take an additional cycle. Das et al. [6] use a substitute cache (SC) to hold critical words. Like SFP's L1-PB, the SC is accessed in parallel with the L1. Unlike SFP, the SC holds words (not blocks) and does not leverage prefetching. Liang and Brooks [15, 16] explore how to mitigate the impact of variability on floating point units (FPUs) and register file read accesses. For pipelined multi-cycle FPUs, they use time borrowing between stages, and they add an extra set of latches that can be used to add an extra stage to the FPU pipe. By adding a stage, they can avoid slowing the clock if variability causes the FPU to be slower than nominal. For the register file, they steer read accesses such that fast entry/port combinations are prioritized. They do not consider register file writes. ReCycle [22] extends the idea of time borrowing across pipeline stages to include the entire pipeline. Time borrowing is orthogonal to our work; if we can mitigate the impact of variability within a structure, we make time borrowing easier by requiring less slack to borrow. In general, though, time borrowing is difficult, because slow stages are vastly more likely than fast stages.

Timing Speculation. Razor flip-flops [7] and X-Pipe flip-flops [24], which were designed for timing speculation (i.e., avoiding worst-case design), can help to mitigate the impact of variability. For flip-flops that could suffer timing problems due to being clocked at the end of a critical or near-critical path, they add a shadow flip-flop that is clocked a half cycle later. If the two flip-flops hold different data, then a timing problem occurred and the processor recovers. However, if a Razor flip-

flop is on a frequently used path that is too slow due to process variability, then the processor will suffer many recoveries and incur a significant performance degradation. Also, the Razor approach cannot be used on circuit paths that could be shorter than half of a clock cycle.

8. CONCLUSIONS

In this paper, we have developed techniques for alleviating the impact of process variability on register files, functional units, and L1 caches. In fact, our schemes enable us to clock processors aggressively instead of being bottlenecked by slow components. We believe that architectural techniques such as these will be crucial for enabling future, more variability-prone CMOS technologies to provide increasing performance.

ACKNOWLEDGMENTS

This material is based on work supported by the National Science Foundation under grants CCF-0444516, CCF-0545456, CCF-0540994, and EIA-9972879, the National Aeronautics and Space Administration under grant NNG04GQ06G, and Intel Corporation. We thank Brian Fields for graciously sharing his criticality predictor with us. We thank Ismet Bayraktaroglu for helpful feedback.

REFERENCES

[1] A. Agarwal et al. Process Variation in Embedded Memories: Failure Analysis and Variation Aware Architecture. *IEEE Journal of Solid-State Circuits*, 40(9):1804–1814, 2005.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[3] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, Nov/Dec 2005.

[4] M. D. Brown, J. Stark, and Y. N. Patt. Select-Free Scheduling Logic. In *Proc. 34rd Annual I Int'l Symp. on Microarchitecture*, pages 204–213, Dec. 2001.

[5] Z. Chishti and T. N. Vijaykumar. Wire Delay is Not a Problem for SMT (in the near future). In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 40–51, June 2005.

[6] A. Das, S. Ozdemir, G. Memik, J. Zambreno, and A. Choudhary. Microarchitectures for Managing Chip Revenues under Process Variations. *IEEE Computer Architecture Letters*, June 2007.

[7] D. Ernst et al. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *Proc. 36th Annual Int'l Symp. on Microarchitecture*, Dec. 2003.

[8] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[9] International Technology Roadmap for Semiconductors, 2003.

[10] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *Proc. 24th Annual Int'l Symp. on Computer Architecture*, pages 252–263, June 1997.

[11] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proc. 17th Annual Int'l Symp. on Computer Architecture*, pages 364–373, May 1990.

[12] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[13] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[14] A.-C. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction & Dead-Block Correlating Prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[15] X. Liang and D. Brooks. Latency Adaptation of Multiported Register Files to Mitigate Variations. In *Proceedings of the Workshop on Architectural Support for Gigascale Integration*, June 2006.

[16] X. Liang and D. Brooks. Mitigating the Impact of Process Variations on Processor Register Files and Execution Units. In *Proc. 39th Annual Int'l Symposium on Microarchitecture*, Dec. 2006.

[17] S. Nassif. Design for Variability in DSM Technologies. In *Proc. of First Int'l Symp. on Quality of Electronic Design*, pages 451–454, Mar. 2000.

[18] S. Ozdemir et al. Yield-Aware Cache Architectures. In *Proceedings of the 39th Annual Int'l Symposium on Microarchitecture*, pages 15–25, Dec. 2006.

[19] B. F. Romanescu, M. E. Bauer, S. Ozev, and D. J. Sorin. VariaSim: Simulating Circuits and Systems in the Presence of Process Variability. Technical Report 2007-3, Department of Electrical and Computer Engineering, Duke University, June 2007.

[20] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the Tenth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[21] S. T. Srinivasan and A. R. Lebeck. Load Latency Tolerance in Dynamically Scheduled Processors. In *Proc. of the 31st Annual International Symposium on Microarchitecture*, pages 148–159, Nov. 1998.

[22] A. Tiwari, S. R. Sarangi, and J. Torrellas. ReCycle: Pipeline Adaptation to Tolerate Process Variability. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[23] J. W. Tschanz et al. Adaptive Body Bias for Reducing Impacts of Die-to-Die and Within-Die Parameter Variations on Microprocessor Frequency and Leakage. *IEEE Journal of Solid-State Circuits*, 37(11):1396–1402, Nov. 2002.

[24] X. Vera, O. Unsal, and A. Gonzalez. X-Pipe: An Adaptive Resilient Microarchitecture for Parameter Variations. In *Proc. of the Workshop on Architectural Support for Gigascale Integration*, June 2006.